AMIGA

Jorge Schimanski

Auf 3½"-Diskette enthalten:
Sämtliche Beispielprogramme des Buches
und nützliche Hilfsprogramme

Heim Verlag

AMIGA

Jorge Schimanski

SPIELE-PROGRAMMIERUNG IN

Assembler

AUF DEM AMIGA

Auf 3½"-Diskette enthalten:
Sämtliche Beispielprogramme des Buches
und nützliche Hilfsprogramme

Heim Verlag

Spieleprogrammierung in Assembler
auf dem Amiga
Jorgo Schimanski
- 1. Auflage - Darmstadt-Eberstadt : April 1991
ISBN 3-928480-02-2

Druck: Druckerei der **Heim OHG**, 6100 Darmstadt-Eberstadt

Inhaltsverzeichnis

Vorwort

Der Amiga ist bekannt für seine grafischen Fähigkeiten, was auch seine Verkaufszahlen belegen. Allerdings wird er noch vom größten Teil der Benutzer als Spielecomputer eingesetzt, obwohl das seine anderen Stärken nicht in geringster Weise mindert, denn gerade das zeichnet ja einen Computer aus. Mit dem Amiga ist man in der Lage Spiele zu programmieren, die absolut realistisch wirken, eben durch seine hervoragenden grafischen Eigenschaften. Natürlich darf die Stereo-Soundqualität nicht vergessen werden.

Den meisten Benutzern reicht es aber nicht mehr, nur zu spielen. Sie wollen vielmehr selbst Spiele nach ihren eigenen Wünschen programmieren. Aber richtig gute Spiele lassen sich nicht über das Betriebssystem schreiben, da sind schon Kenntnisse der Maschinensprache erforderlich. Hat man die Maschinensprache erlernt, fängt das eigentliche Problem erst an, denn das Erlernen der Hardwareprogrammierung ist sehr kompliziert. Genau hier setzt dieses Buch an.

Das Buch enthält eine Sammlung von Grafikroutinen, die einfach über Parameter die Grafikhardware ansteuern. Damit sind keine Hardwarekenntnisse erforderlich. Für die Profis unter ihnen sind die Listings der Routinen gut dokumentiert. Alle in diesem Buch enthaltenen Routinen können frei in eigene Programme eingebaut und vermarktet werden, sind also Public-Domain.

Kapitel 1

Wieso Maschinensprache ?

Die allererste Frage, die man sich stellen sollte, bevor man anfängt ein Spiel zu programmieren ist, in welcher Programmiersprache soll dies geschehen? Dabei ist diese Frage genaugenommen total überflüßig, denn wenn man ein wirklich gutes Spiel programmieren möchte, so geht dies nur in Maschinensprache. Nur in dieser Sprache kann man einen Computer bis ins Letzte ausreizen. Schon alleine von der Rechengeschwindigkeit her steht Maschinensprache an erster Stelle, denn stellen Sie sich mal vor Sie wollen mehrere Objekte gleichzeitig bewegen, Collisionen abfragen, Hintergrundmusik spielen lassen und noch diverse andere Dinge tun. Dabei würde jede andere Sprache kapitulieren. Man kennt ja die kläglichen Versuche von Spielen, die in "C" programmiert wurden...

Deswegen sind alle Routinen und Beispielprogramme in diesem Buch in Maschinensprache geschrieben. Dabei wird die Beherschung dieser Sprache vorausgesetzt. Jedoch werden keinerlei Vorkenntnisse über die Programmierung von Grafik, Sound etc. benötigt. Darauf wird später in den einzelnen Kapiteln eingegangen. Dieses Buch wurde also nicht nur für Profis geschrieben, sondern ist auch ebenso verständlich für Maschinenspracheanfänger.

1.1 Betriebssystem oder Hardware ?

Als nächstes sollte man sich überlegen, ob man die Routinen vom Betriebssystem für sein Spiel verwenden möchte, oder ob eigene Routinen über die Hardware geschrieben werden sollen. Die Entscheidung bleibt natürlich jedem selbst überlassen, jedoch steht außer Frage, das sein Spiel über die Hardware zu programmieren die effektivste Methode ist. Es ist überhaupt nur über die Hardware möglich, ein wirklich gutes Spiel zu programmieren, denn das Amiga-Betriebssystem wurde leider unverständlicherweise in der Sprache "C" geschrieben. Deswegen sind die darin enthaltenen Routinen auch nicht so schnell, daß man sie für ein gutes Spiel verwenden könnte. Jedoch ist der Vorteil auch nicht von der Hand zu weisen, kann man doch leicht und schnell über die Routinen des Betriebssystem verschiedene Dinge programmieren (z.B. ein Fenster öffnen).

Deswegen sind in diesem Buch alle Routinen, bei denen es auf Geschwindigkeit ankommt, über die Hardware programmiert. Nur vereinzelt werden die Routinen des Betriebssystem verwendet, wie z.B. für die Textausgabe oder die Lade- Speichervorgänge.

Kapitel 2

Das Betriebssystem (Kickstart)

Wie schon im vorigen Kapitel erwähnt wurde, ist das Betriebssystem vom Amiga in "C" geschrieben. Und da wir ja unser Spiel über die Harware programmieren wollen, müssen wir das System ausschalten. Denn sonst könnte es zu unerwünschten Nebeneffekten kommen (Guru meditiert), da das gesamte Amiga-System im sogenannten Supervisor-Mode (Überwacher-Modus) läuft. Hingegen läuft unser eigenes Programm im Uservisor-Mode. Dabei wird das User-Programm 50 mal in der Sekunde überwacht. Dieses erledigt das System im Rasterinterrupt. Dieser wird immer dann erzeugt, wenn der Bildschirm aufgebaut wird. Da dieses bekanntlich 50 mal in der Sekunde geschieht (50 Hz), wird unser Programm auch 50 mal überwacht. Schaltet man das Betriebssystem aus, muß beachtet werden, daß nunmehr nicht mehr alle Routinen des Betriebssystems verwendet werden können. Auf alle Routinen, die mit irgendwelchen Message-Strukturen zu tun haben, muß man leider verzichten. Dieses sind z.B. die Fenster-, Screen-, fast alle Exec- und Diskroutinen. Möchte man weiterhin auf diese Routinen zugreifen, so muß vorher das Betriebssystem wieder eingeschaltet werden.

2.1 Betriebssystem an / ausschalten

Da das Betriebssystem seine Überwachungsvorgänge während des Rasterinterrupt durchführt und die Interrupts einfach vom User nach Wunsch an- bzw. ausgeschaltet werden können, werden wir uns diese Tatsache zu Gute kommen lassen. Wir schalten einfach alle Interrupts aus und benutzen nur die, die wir auch wirklich nur für unser Spiel brauchen.

Für das An- und Ausschalten der Interrupts existieren in der Exec-Library zwei Routinen. Für das Ausschalten ist die Disable () -Routine und für das Anschalten die Enable ()-Routine zuständig. Beide Routinen brauchen keine Parameter. Hier nochmal beide Routinen in Kurzform beschrieben:

Library: exec.library
Routine: Disable ()
Offset: -120 = -$78
Parameter: keine
Erklärung: Diese Routine schaltet alle Interrupts aus.

Library: exec.library
Routine: Enable ()
Offset: -126 = -$7e
Parameter: keine
Erklärung: Diese Routine schaltet alle Interrupts, die mit der Routine Disable () ausgeschaltet wurden, wieder ein.

Es muß jedoch darauf geachtet werden, daß, wenn man eigene Interrupts einschaltet, der Wert des Interruptsregisters zwischengespeichert wird, damit später wieder die alten Interrupts eingeschaltet werden können. Außerdem muß die Adresse der jeweiligen Interruptebene, die man benutzen möchte, auch gerettet werden.

Es sollte auch eine kleine Verzögerung eingebaut werden, bevor die Interrupts ausgeschaltet werden. Damit evtl. der Laufwerksmotor noch auslaufen kann. Unbedingt notwendig ist dies zwar nicht, aber so ist wenigstens die Drive-LED nicht die ganze Zeit am leuchten.

Beispiellisting zum Ausschalten des Systems:

```
system_aus:
  move.l #600000,d0          ; Verzögerungswert
motor_aus:                   ; Verzögerungsschleife
  sub.l #1,d0
```

```
  cmp.l #0,d0
  bne motor_aus
;
  move.l 4,a6
  jsr -120(a6)                    ; Disable ()
;
  move.l $6c,oldirqebene          ; alte Interruptebene retten
                                    (Raster-IRQ)
  move.w $dff01c,intena_buf       ; Interruptregister retten
  move.l #NewIrqRoutine,$6c       ; eigene IRQ-Routine einhän-
                                    gen
  move.w #$7fff,$dfff09a          ; alle Interrupts ausschalten
  move.w #$c010,$dff09a           ; z.B.: Copper-IRQ einschal-
                                    ten
  rts
oldirqebene:  dc.l 0              ; Puffer für alte Interruptrou-
                                    tine
intena_buf:  dc.w 0               ; Puffer für alten Wert vom
                                    Interruptreg.
```

Beispiellisting zum Einschalten des Systems:

```
system_an:
  ori.w #$8000,intena_buf         ; Set/Clr - Bit setzen
  move.w #$7fff,$dff09a           ; alle Interrupts sperren
  move.w intena_buf,$dff09a       ; alte IRQs wieder einschalten
  move.l oldirqebene,$6c          ; System-IRQ-Routine wieder
                                    einhängen
  move.l 4,a6
  jsr -126(a6)                    ; Enable ()
  rts
```

Die beiden Routinen können als Unterroutinen in eigene Programme eingebaut werden. Eine genaue Beschreibung der Interruptregister und Ebenen folgt im nächsten Abschnitt.

2.2 Interrupts - Unterbrechungen

Wie schon erwähnt wurde, überwacht das System während des Rasterinterrupts das Userprogramm, also unser eigenes Programm. Der Amiga kennt jedoch noch 13 weitere Interrupts. Es kann ein Interrupt erzeugt werden, wenn z.B. Daten von Diskette gelesen wurden, ein Signal von der Tastatur empfangen wurde oder vom Copper. Die Möglichkeiten sind sehr vielseitig.

Sie werden sich vielleicht fragen, wozu überhaupt Interrupts in einem Spiel? Stellen sie sich mal vor, sie wollen einen Timer in ihr Spiel einbauen. Um eine genaue Zeitausgabe programmieren zu können, müßten sie schon die Timer der Hardware programmieren. Dies ist jedoch recht kompliziert. Einfacher gehts, wenn z.B. ein Rasterinterrupt erzeugt wird. Da dieser bekanntlich ja 50 mal in der Sekunde ausgelöst wird, zählt man einfach im Interruptprogramm eine Variable bis 50 rauf. Jetzt ist 1 Sekunde rum, berechnet die Zeit neu und gibt sie auf den Bildschirm aus.

Außerdem muß für eine ruck- und flackerfreie Animation einfach ein Interrupt erzeugt werden, wie sie in den folgenden Kapiteln noch feststellen werden.

Welche Interrupts nun auftreten dürfen, wird im INTENA-Register festgelegt. Wurde ein Interrupt erzeugt, wird in das Programm verzweigt, dessen Adresse sich in der dazugehörigen Interruptebene befindet. Im Interruptprogramm muß dann als erstes der aufgetretene Interrupt zurückgesetzt werden. Dafür ist das INTREQ-Register zuständig. Außerdem muß das Interruptprogramm mit einem RTE enden.

Da es 14 mögliche Interrupts gibt, aber nur 7 Interruptebenen, müssen sich mehrere Interrupts eine Ebene teilen. Welcher Interrupt wird aber nun zuerst abgearbeitet? Dazu bekommt jeder Interrupt eine Pseudopriorität zugeteilt. Dabei wird der Interrupt mit der höheren Pseudopriorität zuerst abgearbeitet.

Hier nun die Bitbelegung der beiden Register INTENA und INTREQ. Beide Register haben die selbe Belegung.

Registeradressen: INTENA = $DFF09A
INTREQ = $DFF09C

Beide Register können auch gelesen werden. Dazu exestieren folgende Register.

Registeradressen zum Lesen: INTENAR = $DFF01C
INTREQR = $DFF01E

Registerbelegung:

Bit	Name	IE	Funktion
15	SET/CLR		schreiben/lesen
14	INTEN		Interrupts erlauben
13	EXTER	6	CIA-B oder Expansion-Portinterrupt
12	DSKSYN	5	Disksynchronisationswert erkannt
11	RBF	5	Serieller Portinterrupt
10	AUD3	4	Audiokanal 3 gibt Daten aus
9	AUD2	4	Audiokanal 2 ...
8	AUD1	4	Audiokanal 1 ...
7	AUD0	4	Audiokanal 0 ...
6	BLIT	3	Bitteroperation beendet
5	VERTB	3	Beginn der vertikalen Austastlücke
4	COPER	3	Copperprogramm erzeugt Interrupt
3	PORTS	2	CIA-A oder Expansionsport
2	SOFT	1	Software-Interrupt
1	DSKBLK	1	Diskdaten geladen
0	TBE	1	Ausgabepuffer des seriellen Ports leer

Die Bitnummer gibt gleichzeitig die Pseudopriorität an.

An welcher Adresse im Speicher die Interruptebenen (IE) zu finden sind, listet folgende Tabelle auf:

IE	Adresse (in HEX):
1	$64
2	$68
3	$6c
4	$70
5	$74
6	$78

Beschreibung der Registerbelegung:

Will man einen Interrupt erlauben, so muß das SET/CLR-Bit(15), das INTEN-Bit(14) und das entsprechende Interrupt-Bit gesetzt werden. Es soll zum Beispiel der Rasterinterrrupt erlaubt werden. Es müssen also die Bits 15, 14 und 5 im INTENA-Register gesetzt werden:

```
move.w #$c020,$dff09a      ; Bits 15, 14 und 5 setzen
```

Um einen Interrupt zu löschen, muß man Bit 15 und das dazugehörige Interrupt-Bit löschen und alle restlichen Bits setzen. Es sollen zum Beispiel alle Interrupts gelöscht werden:

```
move.w #$7fff,$dff09a      ; alle Bits, bis auf Bit 15 setzen
```

Ist ein Interrupt aufgetreten, so muß dieser vom Interruptprogramm zurückgesetzt werden. Dazu muß lediglich das entsprechende Interrupt-Bit im INTREQ-Register gesetzt werden.

Beispiel zum Zurücksetzen des Rasterinterrupts:

```
move.w #$0020,$dff09c      ; Bit 5 setzen (VERTB)
```

2.3 Copper / Raster- Interrupt

Nach soviel Theorie wollen wir auch mal ein Beispiel zur Interrupterzeugung durchsprechen. Für die Spieleprogrammierung eignet sich dafür am besten der Raster (VertB)- oder der Copper-Interrupt, wobei der Copperinterrupt wohl am besten geeignet ist.

Als erstes wollen wir den Raster bzw. VertikalBlank (VertB) Interrupt besprechen. Um einen solchen Interrupt erzeugen zu können, müssen wir ihn erstmal erlauben. Dazu setzen wir die Bits 15, 14 und 5 im INTENA- Register. Bevor wir jedoch diese Bits setzen, sollte vorher das System ausgeschaltet werden. Wie dieses zu programmieren ist, wurde bereits schon besprochen. Danach sollten alle Interrupts ausgeschaltet werden. Erst jetzt dürfen wir die entsprechenden Bits setzen. Außerdem müssen wir noch die Adresse unser Interruptroutine in die Adresse $6C (Interruptebene 3) eintragen.

In unser Interruptroutine müssen wir dann als erstes das VertB-Bit im INTREQ-Register zurücksetzen und alle Daten- und Adressregister auf dem Stapel gerettet werden, damit das unterbrochende Programm wieder fehlerfrei fortgesetzt werden kann. Als letztes muß die Routine mit einem RTE enden.

Hier nun das Beispiellisting als Unterroutine dazu:

```
Raster_IRQ_ON:
  move.l #600000,d0          ; Verzögerungswert
motor_aus:                   ; Verzögerungsschleife
  sub.l #1,d0                ; Warten bis DiskMotor aus
  cmp.l #0,d0
  bne motor_aus
;
  move.l 4,a6                ; System auschalten
  jsr -120(a6)               ; Disable ()
;
```

```
  move.l $6c,oldirqebene        ; alte Interruptebene retten
                                  (Raster-IRQ)
  move.w $dff01c,intena_buf     ; Interruptregister retten
  move.l #NewIrqRoutine,$6c     ; eigene IRQ-Routine einhän-
                                  gen
  move.w #$7fff,$dff09a         ; alle Interrupts ausschalten
  move.w #$c020,$dff09a         ; Raster-IRQ einschalten
 rts

oldirqebene:  dc.l 0            ; Puffer für alte Interruptrou-
                                  tine
intena_buf:   dc.w 0            ; Puffer für alten Wert vom
                                  Interruptreg.

;--- Ab hier neue Interruptroutine ---

NewIrqRoutine:
  move.w #$0020,$dff09c         ; VertB-Bit für RasterIRQ zu-
                                  rücksetzen
  movem.l d0-d7/a0-a6,-(sp)     ; Daten-Adressregister auf
                                  Stapel retten
                                ; hier steht IRQ-Programm
  movem.l (sp)+,d0-d7/a0-a6     ; Daten-Adressregister vom
                                  Stapel zurück
RTE                             ; Interruptprogramm beenden
```

Das System benutzt, wie sie mittlerweile bestimmt schon wissen, den Rasterinterrupt. Dieser wird ja bekanntlich während der vertikalen Austastlücke erzeugt bzw. bevor ein Bild von der Hardware aufgebaut wird. Für die Spieleprogrammierung ist es aber besser seine Animation ablaufen zu lassen, wenn der Rasterstrahl gerade am unteren Bildschirmrand angekommen ist. Diese Methode bringt einem mehr Zeit und damit eine ruckfreiere Animation.

Wie fragen wir jedoch die Position des Rasterstrahls ab? Eine Möglichkeit wäre, das VHPOS- und VPOS-Register abzufragen (Adressen: $DFF006 und $DFF004) und dann durch Schreiben in das INTREQ-Register einen Copperinterrupt zu erzeugen. Dieses

ist jedoch viel zu unregelmäßig, weil wir das Register von unserem eigenem Programm abfragen müßten. Zudem würde viel zuviel Rechenzeit vergeudet werden. Damit können wir diese Methode schon einmal vergessen.

Die andere Möglichkeit wäre den Copper selbst den Interrupt auslösen zu lassen. Denn mit dem Copper ist man in der Lage auf jede beliebige Rasterzeile zu warten. Wir warten also einfach bis der Rasterstrahl am unterem Bildschirmrand angekommen ist und setzen mit dem Copper das Copper-Bit im INTREQ -Register, denn nur so kann ein Copperinterrupt erzeugt werden. Dazu muß folgender Copperbefehl ins Copperprogramm eingefügt werden:

dc.w $009c,$8010

Wie der Copper programmiert wird, wird im Kapitel 5 beschrieben. Ansonsten ist der Vorgang identisch mit dem der Raster-IRQ-Erzeugung. Es muß lediglich das Bit 5 durch Bit 4 im INTENA- und INTREQ-Register ersetzt werden.

Kapitel 3

Speicherverwaltung

Der Amiga verwaltet seinen Speicher dynamisch. Das heißt, ganz gleich um welche Daten es sich handelt, sie können überall im Speicher abgelegt werden. Dadurch ist es möglich, mehrere Programme im Speicher gleichzeitig ablaufen zu lassen. Deswegen muß man sich einen Speicher vom System zuweisen lassen, bevor man Daten in ihm ablegen kann. Wenn willkürlich Daten im Speicher abgelegt werden würden, könnten zum Beispiel andere Programme gelöscht oder Bilder verunstaltet werden. Dies wiederum wäre ein gefundenes Fressen für den Guru...

Das normale 512 KB CHIP-Ram des Amiga 500/2000 befindet sich von der Adresse $000000 bis $080000. Wobei jedoch die Adressen von $000000 bis $000400 für die Vektoren des Processors reserviert sind. Diese sollte man meiden zu beschreiben. Ansonsten meldet sich der Guru. Für Amiga 500 User, die eine Speichererweiterung von 512 KB besitzen, Amiga 2000 User sowieso, steht noch der Speicherbereich von $C00000 bis $C80000 zur Verfügung. Dabei ist zu beachten, daß sich das System nicht merkt, welcher Speicher belegt wurde, sondern welcher noch zur Verfügung steht. Der Speicher wird über Listen verwaltet, die miteinander Verbunden sind. Aber auch wenn das System, wie in Kapitel 2 beschrieben wurde, ausgeschaltet wird, funktioniert die Speicherverwaltung noch. In dieser Hinsicht brauchen sie also nichts zu befürchten.

Es sollte nach Beendigung eines Programmes sämtlicher von dem Programm reservierter Speicher wieder freigegeben werden. Sonst würde die Speicherkapazität schnell ausgeschöpft sein und nichts paßt dann mehr in den Speicher.

3.1 Unbestimmten freien Speicher reservieren

Zum Reservieren und Freigeben von unbestimmtem freien Speicher, stellt uns die Exec-Library zwei Routinen zur Verfügung. Dabei handelt es sich um die Routinen ALLOCMEM () zum Reservieren und FREEMEM () zum Freigeben des Speichers.

Bevor jedoch Speicher reserviert werden kann, muß dem System mitgeteilt werden, wie groß und um was für eine Art von Speicher es sich handeln soll. Bei der Größe ist zu beachten, daß diese durch acht teilbar ist. Wenn nicht, rundet das System die Größe bis zum nächsten 8-Byte-Schritt auf. Welche Bedingungen gewählt werden können, wird im folgenen beschrieben.

Code	Name	Funktion
$00001	PUBLIC	keine Speicherverschiebung möglich
$00002	CHIP	reserviert Chip-RAM
$00004	Fast	reserviert Fast-RAM
$10000	CLEAR	Speicher wird mit Nullen gefüllt
$20000	LARGEST	reserviere größten Speicherblock

PUBLIC-Bedingung:

Durch diese Bedingung wird festgelegt, daß der zugewiesene Speicher nicht mehr verschoben werden darf. Diese Verschiebung ist in der Kickstart-Version 1.2 nicht enthalten. Man sollte aber dies für spätere Versionen beachten.

CHIP-Bedingung:

Damit wird angegeben, daß es sich bei dem zu reservierenden Speicher um CHIP-Speicher handelt, also in den ersten 512 KB liegt. Dieser Bereich wird von allen DMA-Kanälen benutzt. Deswegen müssen Daten wie zum Beispiel Grafik, Sound und Disk auf

die diese Kanäle zugreifen im CHIP-RAM liegen. Auch wenn sie nur 512 KB Speicher besitzen und somit automatisch CHIP-RAM reserviert wird, geben sie aus Kompatibilitätsgründen diese Bedingung, sofern erforderlich, an.

FAST-Bedingung:

Durch diese Bedingung wird Fast-Speicher reserviert. Dies funktioniert aber nur, wenn eine Speichererweiterung angeschlossen ist.

CLEAR-Bedingung:

Diese Bedingung gibt an, daß der zu reservierende Speicher vor der Zuweisung mit Nullen gelöscht wird.

LARGEST-Bedingung:

Gibt an, daß der größte zur Verfügung stehende Speicherblock reserviert werden soll.

Wird keine Angabe darüber gemacht, ob es sich um CHIP- oder FAST-Speicher handeln soll, wird zuerst versucht, FAST-Speicher zu reservieren, danach erst CHIP-Speicher.

Die Exec-Routinen AllocMem () und FreeMem ():

Routine: AllocMem (D0,D1) (Größe,Code)
Library: exec.library
Offset: -198 = -$c6
Parameter: D0 = Gibt die Größe des Speichers an der reserviert werden soll.
D1 = Bedingungs-Code, um was für eine Art von Speicher es sich handeln soll (siehe oben).

Rückgabe: in D0 = Adresse des reservierten Speicherbereichs. Wenn eine Null zurückgegeben wird, konnte der Speicher nicht reserviert werden.
Erklärung: Diese Routine reserviert einen unbestimmten Speicherbereich von angegebener Größe und Bedingung.

Routine: FreeMem (A1,D0) (Adresse,Größe)
Library: exec.library
Offset: -210 = -$d2
Parameter: A1 = Adresse des Speicherblocks, welcher freigegeben werden soll.
D0 = Gibt die Größe des Speichers an, den man freigeben möchte.
Erklärung: Diese Routine gibt den Speicherbereich wieder frei, welcher zuvor mit der Routine AllocMem () reserviert worden ist.

Manchmal kann es nützlich sein zu wissen, wieviel Speicher einem noch zur Verfügung steht. Dazu stellt uns die Exec-Library die Funktion AvailMem () zur Verfügung:

Routine: AvailMem (D1) (Code)
Library: exec.library
Offset: -216 = -$d8
Parameter: D1 = Bedingungs-Code des Speichers, welcher untersucht werden soll.
Rückgabe: in D0 = Größe des Speichers, der noch zur Verfügung steht.
Erklärung: Diese Routine gibt die Größe des Speichers an, der noch zur Verfügung steht, bezogen auf die Bedingung.

Zur Verdeutlichung hier nun ein Beispiellisting, das eine Speichergröße von 10000 Bytes reserviert. Auf Knopfdruck der linken Maustaste wird der Speicher wieder freigegeben.

```
;--- Zuweisung von Speicher ---
;
  move.l 4,a6                    ; Execbasis
  move.l #10000,d0               ; Speichergröße=10000 Bytes
  move.l #$10002,d1              ; Bedingung: CHIP und
                                   CLEAR
  jsr -198(a6)                   ; AllocMem ()
  move.l d0,speicheradresse      ; Adresse speichern
  tst.l d0                       ; auf Fehler testen
  bne ok
  rts                            ; Programm ende, weil Fehler
ok:
  btst #6,$bfe001                ; Linke Maustaste abfragen
  bne ok
;
  move.l 4,a6                    ; Execbasis
  move.l #10000,d0               ; Speichergröße
  move.l speicheradresse,a1      ; Adresse des Speicherblocks
  jsr -210(a6)                   ; FreeMem ()
rts                              ; Programmende
;
speicheradresse: dc.l 0          ; Puffer zum Speicher der
                                   Adresse
```

3.2 Bestimmten freien Speicher reservieren

Neben der Möglichkeit, sich unbestimmten Speicher zuweisen zu lassen, gibt es noch die der bestimmten Speicherzuweisung. Dadurch ist man in der Lage, sich eine bestimmte Speichergröße ab einer gewünschten Adresse zuweisen zu lassen. Dies ist zum Beispiel nötig, wenn ein Programm "absolut" assembliert wurde. Solch ein Programm kann dann nur an der dazugehörigen Adresse gestartet werden.

Für diese Art der Speicherzuweisung stellt uns die Exec-Library die Funktion AllocAbs () zur Verfügung.

Funktion AllocAbs ():

Routine: AllocAbs (D0,A1) (Größe,Adresse)
Library: exec.library
Offset: -204 = -$cc
Parameter: D0 = Größe des Speicherblocks, der reserviert werden soll.
A1 = Anfangsadresse des Speicherblocks, welcher reserviert werden soll.
Rückgabe: in D0 = Die Adresse, welcher vorher übergeben wurde oder eine Null, wenn der Speicherblock nicht zugewiesen werden konnte.
Erklärung: Diese Routine versucht einen Speicherblock an der angegebenen Adresse und von angegebener Größe zu reservieren. Bei Mißlingen wird in D0 eine Null als Fehlermeldung zurückgegeben, ansonsten die Adresse, die zuvor auch übergeben wurde.

Kapitel 4

Das DOS-System

Ein wichtiger Teil des Amiga-Betriebssystem ist das DOS, was für Disk-Operating-System steht. Über das DOS werden alle Ein- und Ausgabe-Funktionen gesteuert. Wir wollen uns jedoch nur auf die Disketten-Operationen beschränken, wobei wir nur auf die Routinen eingehen werden, mit denen Daten von Diskette gelesen bzw. auf Diskette geschrieben werden können.

Da wir bei den Disketten-Routinen auf das Betriebssystem zurückgreifen, muß vorher natürlich gewährleistet sein, daß das System auch angeschaltet ist. Dieses ist natürlich der Normalfall. Nur wenn Sie später bei der Graphikprogrammierung das System ausschalten, sollten Sie dieses wieder einschalten, bevor Sie z.B. Daten von Diskette laden. Das Ein- und Ausschalten wäre natürlich nicht notwendig, wenn wir unsere eigenen Diskroutinen über die Hardware schreiben würden. Dieses ist jedoch unwahrscheinlich kompliziert und würde den Rahmen des Buches sprengen.

4.1 Die DOS-Library

Die DOS-Library ist die Bibliothek des DOS-Systems, welche alle nötigen Routinen für die Ein- und Ausgabe-Funktionen beinhaltet. Diese Bibliothek befindet sich fest im Speicher und muß, bevor darauf zu gegriffen werden kann, über die Exec-Routine OpenLibrary () geöffnet und sollte nach Beendigung auch wieder geschlossen werden.

Hier ein Beispiellisting zum Öffnen und Schließen der Dos-Library:

```
DOS:
  move.l 4,a6                ; Exec-Basis
  lea dosname,a1             ; Zeiger auf Librarynamen
  clr.l d0                   ; Version = 0
  jsr -552(a6)               ; OpenLibrary ()
  move.l d0,dosbase          ; über diese Basis kann auf
                               die DOS-Routinen zurück-
                               gegriffen werden

  tst.l d0
  bne ok                     ; auf Fehler testen
  rts                        ; wenn Fehler, dann Ende
ok:
 ;
 ; Hier wird DOS-Funktion ausgeführt
 ;
CloseDOS:                    ; DOS-Library wieder schlie-
                               ßen
  move.l 4,a6                ; Exec-Basis
  move.l dosbase,a1          ; DOS-Basis
  jsr -414(a6)               ; CloseLibrary ()
  rts                        ; Ende
;
dosbase: dc.l 0              ; Speicher für DOS-Basis-
                               adresse
dosname:
  dc.b "dos.library",0       ; Dosname in Kleinbuchsta-
                               ben (Wichtig)
  even
```

4.2 Daten von Diskette laden

Für das Laden von Daten von Diskette stellt uns das DOS die Routine READ () zur Verfügung. Ihr braucht man lediglich die Anzahl der Daten und die Adresse, wo diese abgelegt werden sollen, zu übergeben. Allerdings braucht diese Funktion noch zusätzlich die Fileadresse, ab der die Daten von Diskette gelesen werden sollen. Zur Lösung dieses Problems stellt uns das DOS eine weitere Funktion bereit, welche sich OPEN () nennt. Über diese Funktion lassen sich fast alle Ein- Ausgabeoperationen steuern. Wir gehen hier allerdings nur auf das Lesen und Speichern von Daten ein. Für das Lesen muß dieser Funktion in D2 der Wert 1005 und in D1 die Adresse des Filenames, von dem die Fileadresse gewünscht wird, übergeben werden. Damit auch später wieder ordnungsgemäß auf das File zugegriffen werden kann, muß nach Beendigung des Lesevorgangs das File wieder geschlossen werden. Dieses erledigt die DOS-Funktion CLOSE ().

Hier nochmal eine Kurzbeschreibung der eben beschriebenen Routinen:

Routine: OPEN (D1,D2) (Modus,Filename)
Library: dos.library
Offset: -30 = -$1e
Parameter: D1 = Wert 1005 für Lesen der Daten von Disk
D2 = Adresse des Filenames, der mit Null endet
Rückgabe: in D0 = Fileadresse des Files auf Diskette oder Null, wenn File sich File nicht auf Diskette befindet.
Erklärung: Diese Routine gibt die Fileadresse zurück um über ihr weitere Funktionen des DOS steuern zu können (siehe READ).

Routine: READ (D1,D2,D3) (Fileadresse,Datenpuffer,Länge)
Library: dos.library
Offset: -42 = -$2a

Parameter: D1 = Adresse des Files auf Diskette. Liefert uns die Routine OPEN () in D0.
D2 = Adresse des Datenpuffers, in dem die Daten abgelegt werden sollen.
D3 = Anzahl Bytes die geladen werden sollen.
Rückgabe: in D0 = Anzahl Daten, die tatsächlich gelesen wurden. Normalerweise ist diese gleich der, die in Länge übergeben wurde.
Erklärung: Diese Routine lädt eine bestimmte Anzahl Daten von Diskette in den Speicher.

Routine: CLOSE (D1) (Fileadresse)
Library: dos.library
Offset: -36 = -$24
Parameter: D1 = Fileadresse des Files, welches zuvor mit OPEN () geöffnet wurde.
Erklärung: Diese Routine schließt ein File wieder, damit später auch ordnungsgemäß wieder darauf zugegriffen werden kann.

Wie schon erwähnt wurde, muß natürlich die DOS-Library geöffnet sein, bevor auf die Funktionen zugegriffen werden kann.

Das folgende Beispiellisting lädt 100 Bytes eines Files von Diskette. Der Filename kann beliebig gewählt werden, ebenso die Länge.

```
;--- Daten von Diskette laden ---
Read:
  move.l 4,a6                ; ExecBasis
  clr.l d0                   ; Version = 0
  lea dosname,a1             ; Libraryname = DOS
  jsr -552(a6)               ; OpenLibrary ()
  move.l d0,dosbase          ; DOS-Basisadresse speichern

  tst.l d0
```

```
  bne ok                          ; konnte Library geöffnet
                                    werden?
  rts                             ; wenn nicht, Ende
ok:
  move.l #1005,d2                 ; Moduswert = 1005 für Lesen
  move.l #filename,d1             ; Adresse des Filenames
  move.l dosbase,a6               ; DOS-Basisadresse
  jsr -30(a6)                     ; OPEN ()
  move.l d0,filehd                ; Fileadresse speichern
  tst.l d0
  beq exit1                       ; Wenn File nicht auf Disk,
                                    Exit1
  move.l filehd,d1                ; Fileadresse
  move.l #datenpuffer,d2          ; Puffer für Daten
  move.l #100,d3                  ; 100 Bytes lesen
  move.l dosbase,a6               ; Dos-Basisadresse
  jsr -42(a6)                     ; READ ()
  move.l filehd,d1                ; Fileadresse
  move.l dosbase,a6               ; Dos-Basisadresse
  jsr -36(a6)                     ; CLOSE ()
exit1:
  move.l 4,a6                     ; ExecBasis
  move.l dosbase,a1               ; Dos-Basisadresse
  jsr -414(a6)                    ; CloseLibrary ()
  rts
;
;--- Parameter ---
dosbase: dc.l 0                   ; Puffer für DOS-Basisadres-
                                    se
dosname: dc.b "dos.library",0     ; Name der Library
  even                            ; gerade Adresse
filename:                         ; Filename
  dc.b "c/dir",0
  even
filehd: dc.l 0                    ; Puffer für Fileadresse
datenpuffer: blk.b 100,0          ; Puffer in dem die Daten ab-
                                    gelegt werden
;--- Listingende ---
```

4.3 Daten auf Diskette speichern

Für das Schreiben von Daten auf Diskette stellt uns das DOS die Routine WRITE () zur Verfügung. Ihr wird in D3 die Anzahl der Bytes und in D2 die Adresse, ab der die Daten liegen, übergeben. Natürlich ist auch hier, wie bei der READ () - Funktion, eine Fileadresse notwendig. Woher sollte sonst das DOS wissen, auf welche Tracks die Daten geschrieben werden sollen? Diese bekommen wir wiederum von der Funktion OPEN () geliefert, allerdings muß statt dem Moduswert 1005 der Wert 1006 übergeben werden. Nach Beendigung des Schreibvorganges sollte das File wieder mit der Funktion CLOSE () geschlossen werden.

Zum besseren Verständnis der beiden Funktion folgt wiederum eine Kurzbeschreibung:

Routine: OPEN (D1,D2) (Filename,Modus)
Library: dos.library
Offset: -30 = -$1e
Parameter: D1 = Filename des Files
D2 = Wert 1006 für Daten schreiben
Rückgabe: in D0 = Fileadresse des Files auf Diskette oder Null bei Fehler.
Erklärung: Diese Routine liefert die Fileadresse eines Files auf Diskette zurück bzw. eine Null als Fehlermeldung.

Routine: WRITE (D1,D2,D3) (Fileadresse,Datenadresse,Länge)
Library: dos.library
Offset: -48 = -$30
Parameter: D1 = Fileadresse, die wo über die Funktion OPEN () erhalten.
D2 = Anfangsadresse ab der die Daten im Speicher beginnen.
D3 = Anzahl Bytes, die auf Diskette geschrieben werden sollen.
Erklärung: Diese Routine schreibt eine bestimmte Anzahl von Bytes auf Diskette.

Das entsprechende Beispiellisting darf natürlich auch nicht fehlen. Es ist im Prinzip wie das Listing für das Laden von Daten aufgebaut, nur daß in diesem Fall 100 Bytes auf Diskette geschrieben werden.

Achtung! Sie sollten bei der Auswahl des Filenames darauf achten, das dieser noch nicht auf der Diskette existiert. Ansonsten wird das alte File auf der Diskette überschrieben.

```
;--- Daten auf Diskette schreiben ---
Write:
  move.l 4,a6                    ; ExecBasis
  clr.l d0                       ; Version = 0
  lea dosname,a1                 ; Libraryname = DOS
  jsr -552(a6)                   ; OpenLibrary ()
  move.l d0,dosbase              ; DOS-Basisadresse speichern

  tst.l d0
  bne ok                         ; konnte Library geöffnet werden?
  rts                            ; wenn nicht, Ende
ok:
  move.l #1006,d2                ; Moduswert = 1006 für Schreiben
  move.l #filename,d1            ; Adresse des Filenames
  move.l dosbase,a6              ; DOS-Basisadresse
  jsr -30(a6)                    ; OPEN ()
  move.l d0,filehd               ; Fileadresse speichern
  tst.l d0
  beq exit1                      ; Wenn Fehler, Exit1
  move.l filehd,d1               ; Fileadresse
  move.l #datenadresse,d2        ; Adresse der Daten
  move.l #100,d3                 ; 100 Bytes schreiben
  move.l dosbase,a6              ; Dos-Basisadresse
  jsr -48(a6)                    ; WRITE ()
  move.l filehd,d1               ; Fileadresse
  move.l dosbase,a6              ; Dos-Basisadresse
  jsr -36(a6)                    ; CLOSE ()
```

```
exit1:
  move.l 4,a6                    ; ExecBasis
  move.l dosbase,a1              ; Dos-Basisadresse
  jsr -414(a6)                   ; CloseLibrary ()
  rts
 ;
 ;--- Parameter ---
dosbase: dc.l 0                  ; Puffer für DOS-Basisadres-
                                   se
dosname: dc.b "dos.library",0    ; Name der Library
  even                           ; gerade Adresse
filename: dc.b "Demo",0          ; Filename
  even
filehd: dc.l 0                   ; Puffer für Fileadresse
datenadresse: blk.b 100,0        ; Puffer in dem die Daten lie-
                                   gen
;
;--- Listingende ---
```

Kapitel 5

Darstellung von Bildern (Screens)

Der Amiga steuert die Bildschirmausgabe über sogenannte Playfields. Ein Playfield wiederum besteht aus mindestens einer Bitmap bzw. maximal sechs Bitmaps. Eine Bitmap besitzt die Größe eines darzustellenden Bildes, also eines Screens. Besitzt der Screen zum Beipiel eine Breite von 320 und eine Höhe von 256, so ist die Bitmap genauso groß, denn jeder Pixel (Punkt) auf dem Screen spiegelt sich im Speicher wieder.

Sie werden sich sicherlich fragen, wieso wir maximal 6 Bitmaps für einen Screen verwenden können, wo doch eine Bitmap genauso aufgebaut ist wie ein Screen. Mit einer Bitmap kann man nur die Zustände Punkt gesetzt oder gelöscht festlegen. Woher soll nun der Amiga wissen, in welcher Farbe dieser dargestellt werden soll? Dazu legt man einfach mehrere Bitmaps gleicher Größe übereinander. So stehen einem max. 32 Farben bei fünf übereinanderliegenden Bitmaps zur Verfügung. Die Anzahl Farben läßt sich nach folgender Formel berechnen: Anzahl Farben = 2 hoch Bitmapanzahl. Zum Beipiel: 3 Bitmaps = 2 hoch 3 = 8 Farben

Der Amiga bietet einem eine recht große Auswahl an Möglichkeiten, Playfields darzustellen:

- Farbenanzahl zwischen 2 und 4096 gleichzeitig
- verschiedene Auflösungen
- zwei unabhängige Playfields
- Scrolling

Wir werden hier allerdings nur auf die Programmierung eines einfachen Playfields eingehen, denn für die Spieleprogammierung reicht dieses völlig aus. Wir werden hier auch nicht auf die komplizierte Beschreibung der Hardwareregister eingehen. Dafür werden in den weitere Abschnitten Routinen vorgestellt, die uns diese Arbeit abnehmen. Dabei handelt es sich nicht um Systemfunktionen, sondern um von mir selbstgeschriebene Routinen, die die Hardware direkt ansteuern.

5.1 Die Bitmap

Das wichtigste Element zur Darstellung eines Bildes ist wohl die Bitmap, denn sie enthält die Informationen über Form und Größe des Bildes. Diese Informationen werden in Hardwareregistern gespeichert, damit der Amiga weiß, wie das Bild auszusehen hat und wo es zu finden ist. Diese Methode hat leider den Nachteil, daß man selbst nicht mehr auf diese Informationen zugreifen kann. Denn in der Regel sind die Hardwareregister nur Schreibregister, es ist also nur möglich, einen Wert zu speichern, aber nicht zu lesen.

Damit man aber selbst auch noch auf diese Informationen zugreifen kann, legt man eine Struktur an, in der alle nötigen Daten zur Darstellung eines Screens enthalten sind. Diese Struktur wird, sinnvollerweise, BitMap-Struktur genannt.

Aufbau der Bitmap-Struktur: (Länge = 40 Bytes)

Offset	Typ	Bezeichnung	Funktion
00	Word	BytePerRow	Anzahl Bytes einer Zeile
02	Word	Rows	Anzahl Zeilen der Bitmap
04	Byte	Flags	System-Flags
05	Byte	Depth	Anzahl der Bitmaps (Tiefe)
06	Word	Pad	unbenutzt
08	Long	PlanePtr1	Adresse der 1. Bitmap
12	Long	PlanePtr2	Adresse der 2. Bitmap
16	Long	PlanePtr3	Adresse der 3. Bitmap
20	Long	PlanePtr4	Adresse der 4. Bitmap
24	Long	PlanePtr5	Adresse der 5. Bitmap
28	Long	PlanePtr6	Adresse der 6. Bitmap (HAM-Mode)
32	Long	PlanePtr7	Adresse der 7. Bitmap
36	Long	PlanePtr8	Adresse der 8. Bitmap
40		END	Ende der BitMap-Struktur

Beschreibung:

BytePerRow: (Offset 00)

Enthält die Anzahl Bytes, die eine Zeile einer Bitmap lang ist. Ist das Bild zum Beispiel 320 Pixel breit, so beträgt die Anzahl Bytes = 320/8 = 40 Bytes.

Rows: (Offset 02)

Enthält die Höhe der BitMap. Ist das Bild zum Beispiel 256 Zeilen hoch, so steht hier der Wert 256.

Flags: (Offset 04)

Enthält einige Bitinformationen über die Darstellung. Für den User unwichtige Informationen.

Depth: (Offset 05)

Enthält die Anzahl Bitmaps, die der Screen besitzt. Die Anzahl richtet sich nach der Anzahl der verwendeten Farben. Bei 32 Farben steht hier der Wert 5.

PlanePtrX: (Offset 08 bis 36)

Hier stehen die Adressen der einzelnen Bitmaps. Die PlanePtr 7 und 8 sind für spätere Erweiterungen vorgesehen. Momentan werden nur die ersten sechs benutzt. Dabei wird die sechste für den HAM-Modus verwendet.

Um nun die BitMap-Struktur mit den entsprechenden Werten zu füllen, kann man dieses entweder von Hand machen, was jedoch sehr umständlich ist, oder man verwendet die System-Fuktion "InitBitMap ()" aus der graphics.library. Jedoch muß ber der System-Funktion erst umständlich die graphics.library geöffnet werden und außerdem müssen danach die PlanePtrX-Adressen noch nachträglich eingetragen werden. Zudem ist diese Routine viel zu langsam. Deswegen stelle ich ihnen hier eine selbstgeschriebene Routine vor, die kompatibel zum System ist, aber einige Funktionen mehr enthält und außerdem schneller ist. Diese Routine finden sie, wie alle anderen in diesem Buch, auf der beiliegenden Diskette.

Routine: InitBitMap (A0,D0,D1,D2,D3)
(BitMap,Depth,Width,Height,Memory)

Parameter: A0 = Zeiger auf BitMap-Struktur (Puffer von 40 Bytes)
D0 = Anzahl Bitmaps (Tiefe)
D1 = Breite in Pixel - Achtung: muß durch 16 teilbar sein
D2 = Höhe der Grafik
D3 = 1, dann wird auch gleich Speicher für PlanePtr reserviert und Adressen in Struktur eingetragen
D3 = 0, kein Speicher wird für PlanePtr reserviert

Rückgabe: D0 = 0 , alles OK.
D0 = -1, Breite konnte nicht durch 16 geteilt werden
D0 = -2, Speicher für PlanePtr konnte nicht reserviert werden

Erklärung: Initialisiert eine BitMap-Struktur mit den entsprechenden Werten, genau wie die gleichnamige System-Routine. Nur ist diese Routine schneller, und außerdem kann man, wenn man D3 = 1 setzt, automatisch gleich Speicher für die BitMap-Pointer reservieren lassen. Diese werden sogleich in die Struktur gespeichert.

```
;--- BitMap-Struktur initialisieren und Speicher für Maps    ---
;--- reserverieren                                           ---
;--- D0 = Depth, D1 = Width, D2 = Height                     ---
;--- D3 = Memory, A0 = BitMap                                ---

InitBitMap:
  move.w d1,d4          ; Breite nach D4
  and.w #15,d4          ; Rest ermitteln
  tst.w d4              ; Rest vorhanden ?
  beq ibm_1             ; Wenn nicht, dann weiter -
                          sonst Error
  move.l #-1,d0         ; Error: Breite nicht durch 16
                          teilbar
  rts
ibm_1:
  lsr.w #4,d1           ; Breite durch 16 teilen
  lsl.w #1,d1           ; mal 2 = Anzahl Bytes einer
                          Zeile
  move.w d1,(a0)        ; und in BitMap-Struktur
                          speichern
  move.w d2,2(a0)       ; Höhe in BitMap-Struktur
                          speichern
  move.w d0,4(a0)       ; Anzahl Planes speichern
  tst.w d3              ; Muß Speicher für BitMaps
                          reserviert werden ?
```

```
  bne ibm_2                    ; Wenn nicht, dann Ende,
                                 sonst weiter
  clr.l d0                     ; No Errors
  rts                          ; Ende
ibm_2:
  move.w d0,d4                 ; Depth nach D4
  sub.w #1,d4                  ; minus 1
  move.l a0,a1                 ; BitMap nach a1
  add.l #8,a1                  ; Anfang BitMapPointers
ibm_clear_loop:                ; Pointer-Zeiger
  clr.l (a1)+                  ; löschen
  dbra d4,ibm_clear_loop
  mulu d2,d1                   ; BytePerRow x Höhe = Size
                                 von einer Map
  move.w d0,d2                 ; D2 = Anzahl Planes
  move.l d1,d0                 ; D0 = ByteSize
  move.l #$10002,d1            ; D1 = CHIP + FreeMem
  sub.w #1,d2                  ; Anzahl Planes minus 1, we-
                                 gen DBRA
  add.l #8,a0                  ; Anfang BitMap-Zeiger
  move.l a0,a2                 ; auch nach A2
ibm_loop:
  move.l 4,a6
  movem.l d0-d2/a0-a2,-(sp)
  jsr -198(a6)                 ; AllocMem
  tst.l d0                     ; konnte Speicher reserviert
                                 werden ?
  bne ibm_l1                   ; Wenn ja, dann weiter
  movem.l (sp)+,d0-d2/a0-a2
ibm_free_loop:
  move.l (a2),a1               ; Zeiger auf BitMap holen
  cmp.l #0,a1                  ; Null ?
  bne ibm_l2                   ; wenn ja, dann Ende
  move.l #-2,d0                ; Error
  rts                          ; Ende
```

```
ibm_l2:
  clr.l (a2)+
  movem.l d0/a2,-(sp)
  move.l 4,a6
  jsr -210(a6)                    ; FreeMem
  movem.l (sp)+,d0/a2
  bra ibm_free_loop
ibm_l1:
  move.l d0,d5                    ; Zeiger auf BitMap nach D5
  movem.l (sp)+,d0-d2/a0-a2
  move.l d5,(a0)+
  dbra d2,ibm_loop
  clr.l d0
  rts
```

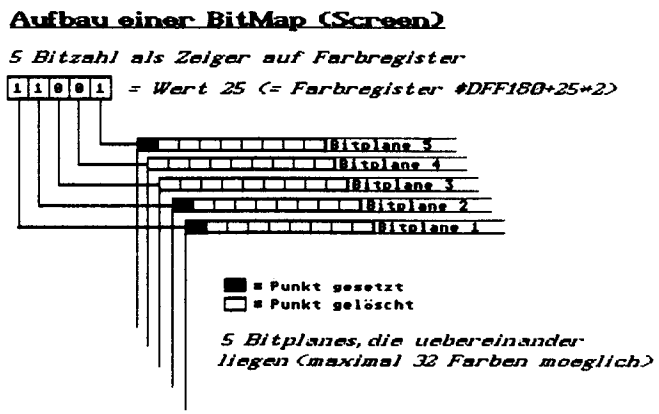
Diese Routine bedarf wohl keiner näheren Erläuterung, da sie ausreichend dokumentiert ist.

Wenn die BitMaps nicht mehr gebraucht werden, zum Beispiel bei Beendigung des Programms, sollte der von ihnen belegte Speicher auch wieder freigegeben werden. Dies kann natürlich durch Berechnen mit den entsprechenden Werten aus der BitMap-Struktur jeder selbst programmieren. Das wäre jedoch wieder viel zu umständlich. Deswegen stelle ich ihnen die folgende Routine vor, die diese Aufgabe sehr komfortabel erledigt. Der Routine habe ich den Namen "ClearBitMap()" gegeben.

Beschreibung der Funktion ClearBitMap ():

Routine: ClearBitMap (A0) (BitMap)
Parameter: A0 = Adresse der BitMap-Struktur
Erklärung: Gibt den mit InitBitMap () reservierten Speicher der PlanePtr wieder zurück und löscht danach diese Zeiger.

```
;--- Speicher der Maps wieder freigeben in einer
;--- BitMap-Struktur
;--- A0 = BitMap
ClearBitMap:
  clr.w d1
  move.b 5(a0),d1                ; Depth nach D1
  sub.w #1,d1
  move.w (a0),d0                 ; BytePerRow
  mulu 2(a0),d0                  ; mal Höhe = MapSize
  add.l #8,a0
cbm_loop:
  move.l (a0),a1                 ; Map-Pointer
  cmp.l #0,a1
  beq cbm_l1
  clr.l (a0)+
  movem.l d0-d1/a0,-(sp)
  move.l 4,a6
  jsr -210(a6)                   ; FreeMem
  movem.l (sp)+,d0-d1/a0
cbm_l1:
  dbra d1,cbm_loop
  rts
```



5.2 IFF-Bilder ausgeben

Sie kennen bestimmt das Malprogramm Deluxe-Paint. Wenn sie mit diesem Malprogramm Bilder zeichnen und diese dann auf Diskette abspeichern, kann es von fast jedem anderen Malprogramm geladen und weiterbearbeitet werden.

Das liegt daran, daß all diese Zeichenprogramme die Bilder im sogenannten IFF-Standart abspeichern. Dieser "Interchange File Format" Standart (IFF), steht für Datenaustausch zwischen verschiedenen Programmen und wurde vom Software Haus Electronic Arts entwickelt. Das Prinzip ist eigentlich ganz einfach: Es wird zu der eigentlichen Datei (zum Beispiel das Bild) noch zusätzlich ein Header mitabgespeichert, welcher die Datei bis ins kleinste Detail beschreibt (Höhe, Breite etc.). Den IFF-Standart gibt es nicht nur für Grafik, sondern auch für Musik, Text usw. Wir wollen hier aber nicht das komplexe IFF-System erläutern. Damit aber auch sie ihre selbsterstellten Bilder darstellen können, stelle ich ihnen eine Routine vor, die diese Aufgabe bewältigt.

Als erstes müssen Sie das gewünschte Grafikbild in den Speicher laden. Danach programmieren sie einen Screen (siehe Abschnitt 5.9), der genauso groß ist, wie ihr erstelltes Bild. Nun tragen sie die Adresse der Bitmap-Struktur von ihrem Screen in A0 und die Adresse des Bildes in A1 ein. Außerdem lassen sie das Adressregister A2 auf einen Puffer von 64 Bytes zeigen. Jetzt rufen sie die Routine "PrintIffPic ()" auf. Sofort wird ihr erstelltes Bild in die übergebene BitMap gezeichnet. Allerdings in den falschen Farben. Die richtigen Farben finden sie jetzt in den 64 Bytes großen Puffer, welchen sie vorher in A2 übergeben haben. Einfacher kann man wohl ein IFF-Bild nicht darstellen. Ein Beispiel dazu finden sie im Abschnitt 5.9.

Hier noch einmal die Kurzbeschreibung der Routine PrintIffPic ():

Routine: PrintIffPic (A0,A1,A2) (BitMap,IffPic,ColorTable)
Parameter: A0 = Adresse der BitMap-Struktur, in welcher das Bild dargestellt werden soll.
A1 = Adresse des Iff-Bildes im Speicher.
A2 = Adresse eines Puffers von 64 Bytes, in dem die Farben vom Iff-Pic gespeichert werden.
Erklärung: Diese Routine stellt ein Iff-Bild in der gewünschten BitMap da. Dabei ist zu beachten, daß die Breite, Höhe und Tiefe der BitMap-Struktur genausogroß sind, wie das IFF-Bild.

Es folgt das Listing mit Dokumentation der PrintIffPic-Routine:

```
;--- PrintIffPic (A0,A1,A2) (BitMap,IffPic,ColorTable)        ---
;--- zeichnet ein Iff-Pic in die übergebene BitMap            ---
PrintIffPic:
;
  movem.l a0-a2,-(sp)              ; Parameter retten
;
;--- Iff-Chunk-Adressen suchen ---
  move.l a1,pic_buf                ; IffPic-Adresse speichern
  move.l #chunktabelle,a1
  move.l #chunkadresse,a2
PIsearch0:
  move.l pic_buf,a0
  clr.w d0
PIsearch1:
  cmp.l #0,(a1)
  beq PIsearch2
  move.b (a0)+,iffchunk
  move.b (a0)+,iffchunk+1
  move.b (a0)+,iffchunk+2
  move.b (a0)+,iffchunk+3
  move.l iffchunk,d6
  move.l (a1),d7
  cmp.l d7,d6
```

```
  beq Plsearch3
  sub.l #3,a0
  bra Plsearch1
Plsearch00:
  add.l #4,a1
  add.l #4,a2
  bra Plsearch0
Plsearch3:
  sub.l #4,a0
  move.l (a2),a4
  move.l a0,(a4)
  bra Plsearch00
Plsearch2:
  clr.w d0
  move.l bmhd_chunk,a0
  move.b 16(a0),d0
  sub.w #1,d0
  move.w d0,planes_num          ; Anzahl Planes
  move.w 8(a0),width            ; Breite
  move.w 10(a0),height          ; Höhe
;
  movem.l (sp)+,a0-a2           ; Parameter wiederholen
;
;--- PlanePointer-Adressen errechnen ---
  add.l #8,a0                   ; Zeiger auf 1. Bitmappointer
  move.l #planes,a1             ; Bitmappointer-Puffer
  move.w planes_num,d0          ; Anzahl Bitmaps
Pl_loop:
  move.w d0,d1
  mulu #4,d1
  move.l (a0,d1),(a1,d1)
  dbra d0,Pl_loop
;
;--- Iff Farben errechnen und speichern ---
  move.l cmap_chunk,a0
  add.l #8,a0
  move.l a2,a1                  ; Farbenpuffer nach A1
  move.w #31,d7                 ; 32 Farben
```

```
Plco_loop1:
  clr.w d0
  move.b (a0)+,d0
  and.b #$f0,d0
  lsl.w #4,d0
  move.b (a0)+,d0
  and.w #$0ff0,d0
  clr.w d1
  move.b (a0)+,d1
  lsr.b #4,d1
  and.b #$0f,d1
  or.b d1,d0
  move.w d0,(a1)+
  dbra d7,Plco_loop1
;
;---- Parameter für Entpacker ----
  move.l #planes,bitplanezeiger   ; Puffer von PlanePtr-Adressen
  move.l bmhd_chunk,a0            ; BMHD-Chunk-Adresse
  move.b 17(a0),maske             ; MaskenByte setzen
  move.b 18(a0),compression       ; PackerByte
  clr.w d0
  move.b 16(a0),d0                ; Depth
  move.w d0,anzahlplanes          ; speichern
  move.w 8(a0),picbreite          ; Picbreite
  move.l body_chunk,a0            ; BODY-Chunk-Adresse
  add.l #4,a0                     ; plus 4
  move.l (a0)+,piclänge           ; = Piclänge
  move.l a0,picadresse            ; A0 = Pic-Anfangsadresse
  bsr Pl_decrunch                 ; Pic entpacken
  rts                             ; ENDE
;
;--- Iff-Unpacker-Routine ---
Pl_decrunch:
  clr.l d0
  move.w picbreite,d0             ; Picbreite nach D0
  lsr.w #3,d0                     ; durch 8 teilen
  move.w d0,width_bytes           ; und speichern
```

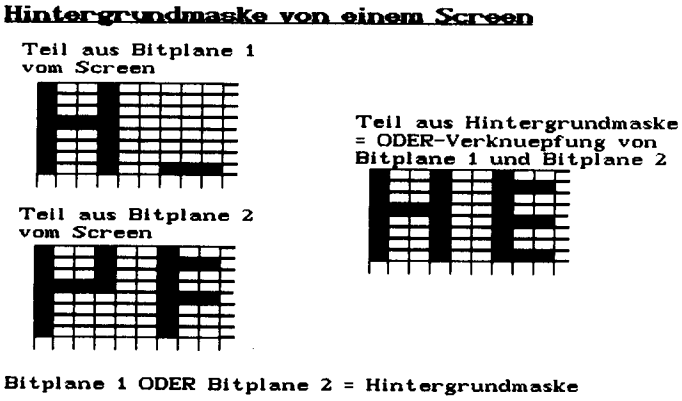
```
  move.w anzahlplanes,d2        ; Depth nach D2
  move.l bitplanezeiger,a2      ; Pufferadresse v. PlanePtr
  move.l picadresse,a0          ; Picadresse nach A0
  move.l a0,a3                  ; nach A3
  add.l piclänge,a3             ; plus Piclänge = Picende
Plunp_loop:
  cmp.l a3,a0                   ; Bild schon entpacked ?
  bge Plunpack_end              ; wenn ja, dann Ende
  clr.w d3                      ; BitMapzähler
Plpic_loop1:
  move.w d3,d4                  ; D3 nach D4
  lsl.w #2,d4                   ; mal 4
  move.l (a2,d4),a5             ; Zeiger auf Bitmap errechnen
  bsr Plunpack_row              ; und Daten entpacken
  move.l a5,(a2,d4)             ; BitMapposition speichern
  addq.w #1,d3                  ; BitMapzähler plus 1
  cmp.w d2,d3                   ; schon eine Reihe aller Maps
                                  durch ?
  blt Plpic_loop1               ; wenn nicht, dann weiter
  andi.b #1,maske               ; sonst Maske entpacken
  beq Plunp_loop                ; wenn vorhanden ?
  move.l #mask_dummy,a5         ; Maskenpuffer
  bsr Plunpack_row              ; entpacken
  bra Plunp_loop                ; wieder von vorne
;
Plunpack_row:
  move.l d2,-(sp)               ; Depth-Variable retten
  move.w width_bytes,d2         ; Zeilenbreite in Bytes
Plunp_loop1:
  tst.w d2                      ; schon eine Zeile durch ?
  beq Plunpack_row_end          ; Wenn ja, dann nächste
  clr.w d0
  tst.b compression             ; Pic gepacked ?
  bne Plunp_comp                ; wenn ja, dann Unp_Comp
  move.w width_bytes,d0         ; sonst Zeilenbreite nach D0
  subq.w #1,d0                  ; minus 1
  bra Plunp_loop2               ; und Bytes übernehmen
Plunp_comp:
```

```
  move.b (a0)+,d0          ; Befehlsbyte holen
  bmi Plpacked             ; Wenn negativ, dann gepakked

Plunp_loop2:
  move.b (a0)+,(a5)+       ; sonst Bytes normal übernehmen

  subq.w #1,d2             ; Zeilenbreite minus 1
  dbra d0,Plunp_loop2      ; Schleife bis Anzahl Null
  bra Plunp_loop1          ; Von vorne
Plpacked:
  neg.b d0                 ; Befehlsbyte negieren
  move.b (a0)+,d1          ; Füllbyte nach d1
Plunp_loop3:
  move.b d1,(a5)+          ; Zeile mit
  subq.w #1,d2             ; Füllbyte beschreiben
  dbra d0,Plunp_loop3      ; Schleife
  bra Plunp_loop1          ; vor vorne
Plunpack_row_end:
  move.l (sp)+,d2          ; Depth-Variable wiederholen
Plunpack_end:
  rts                      ; Ende
;--- Parameter für Entpacker ---
maske: dc.w 0
compression: dc.w 0
anzahlplanes: dc.w 0
bitplanezeiger: dc.l 0
picbreite: dc.w 0
piclänge: dc.l 0
picadresse: dc.l 0
;--- Parameter für Programm ---
chunktabelle: dc.l "BMHD","CMAP","BODY",0
chunkadresse: dc.l bmhd_chunk,cmap_chunk,body_chunk
iffchunk: dc.l 0
;
bmhd_chunk: dc.l 0
cmap_chunk: dc.l 0
body_chunk: dc.l 0
;
```

```
planes_num: blk.w 1,0
width_bytes: blk.w 1,0
;
width: dc.w 0
height: dc.w 0
planes: blk.l 10,0
pic_buf: blk.l 1,0
mask_dummy: blk.b 128,0
;
;--- Listingende der Routine PrintIffPic () ---
```

5.3 Hintergrundmaske

Stellen sie sich mal vor, sie bewegen ein Raumschiff durch eine Felsenlandschaft und bei Berührung explodiert es. Wie kann man nun so etwas programmieren? Die einfachste Methode ist, von der Landschaft bzw. von dem Grafikbild eine Schattenmaske anzulegen. Eine Schattenmaske ist genausogroß wie eine Bitmap des Grafikbildes (Screen). In diesem Puffer steht das Ergebnis einer Oder-Verknüpfung aller Bitmaps eines Screens. Das ist ja auch logisch, denn wenn auch nur in einer Bitmap ein Punkt gesetzt ist, tritt an dieser Stelle eine Collision bei Berührung ein.



Diese Schattenmaske (englisch = ShadowMask) kann man jetzt natürlich ganz umständlich über eine Oder-Verknüpfung in Assembler programmieren. Diese Methode wäre aber viel zu umständlich und zudem noch viel zulangsam. Einfacher und schneller geht es mit dem Blitter. Der Blitter ist ein Co-Prozessor, mit dem ein Datentransfer zwischen maximal 3 Quellen, welche noch auf verschiedene Arten miteinander verknüpft werden können, schnell durchgeführt werden kann. Allerdings ist die Programmierung des Blitters recht kompliziert.

Damit aber auch wir unsere Schattenmaske einfach und blitzschnell erstellen können, stelle ich Ihnen eine Routine vor, die, wie sollte es auch anders sein, diesen Zweck erfüllt. Es handelt sich dabei um die Routine "GetBackMask ()". Dieser Routine braucht man in A0 nur die Anfangsadresse der BitMap-Struktur vom entsprechenden Screen zu übergeben und in A1 die Adresse eines 32 Bytes großen Puffers. Der 32 Bytes große Puffer beinhaltet eine ObjektArgs-Struktur. Auf diese Struktur wird im Kapitel 7 eingegangen. Sie wird für die Darstellung von Grafikobjekten benutzt.

Damit ist man in der Lage, kinderleicht Collisionen zwischen einem Objekt und dem Hintergrund festzustellen. Die Collisionsroutine, die dieses möglich macht, wird auch im Kapitel 7 beschrieben.

Kurzbeschreibung der Routine "GetBackMask ()":

Routine: GetBackMask (A0,A1) (BitMap,ObjektArgs)
Parameter: A0 = Zeiger auf BitMap-Struktur von welcher die Schattenmaske angelegt werden soll.
A1 = Anfangsadresse eines 32 Bytes großen Puffers. Er enthält die ObjektArgs-Struktur, welche im Kapitel 7 beschrieben wird.
Rückgabe: D0 = 0 , Funktion erfolgreich ausgeführt.
D0 = -1, es konnte nicht genug Speicher für die Schattenmaske reserviert werden.

Erklärung: Füllt die in A1 übergebene ObjektArgs-Struktur mit den wichtigsten Werten und reserviert Speicher für die Schattenmaske (ShadowMask/CollMask). Diese ObjektArgs-Struktur kann dann ganz normal mit der Collision () -Routine (Kapitel 7) zur Collision mit anderen Objekten herangezogen werden.

Tip: Wenn man die BitMap-Pointer und die Tiefe in der BitMap-Struktur vorher entsprechend manipuliert, kann man sogar Collisionen nur zwischen bestimmten Farbregistern zulassen.

Listing der Routine "GetBackMask ()":

```
;--- ShadowMask/CollMask vom Hintergrund init.      ---
;--- A0 = BitMap ; A1 = ObjektArgs (32 Bytes)       ---
GetBackMask:
  move.w (a0),d0                ; Byte per Row nach D0
  mulu 2(a0),d0                 ; mal Höhe = MapSize
  move.l #$10002,d1             ; Chip + ClearMEM
  movem.l a0-a1,-(sp)
  move.l 4,a6
  jsr -198(a6)                  ; AllocMem
  movem.l (sp)+,a0-a1
  tst.l d0
  bne gbm_1
  move.l #-1,d0                 ; Error: nicht genug Memory
  rts
gbm_1:
  move.l d0,28(a1)              ; CollMask in Objektargs
                                  speichern
  move.b #1,5(a1)               ; Init = 1
  clr.l 6(a1)                   ; x,y pos = Null
  move.w 2(a0),10(a1)           ; Height
  move.w (a0),d1                ; Byte per Row
  lsr.w #1,d1
  move.w d1,12(a1)              ; WordWidth
  move.l d0,a2                  ; Ziel D und Quelle B nach A2
```

```
  move.w #$0dfc,$dff040          ; BLTCON0: A + B = D
  clr.w $dff042                  ; BLTCON1:  no  Shift-Quelle
                                   B
                                 ; und aufsteigende Adressen
  move.w #$ffff,$dff044          ; First Mask
  move.w #$ffff,$dff046          ; Last Mask
  clr.w $dff064                  ; Modulowert von Quelle A
  clr.w $dff062                  ; Modulowert  von  Quelle  B  =
                                   1 Word
  clr.w $dff066                  ; Modulowert  von  Ziel   D  =  1
                                   Word

  clr.l d5
  move.w (a0),d5                 ; Breite in Bytes
  lsr.w #1,d5                    ; jetzt in Words
  move.w 2(a0),d6                ; Höhe in Pixel
  and.w #$3ff,d6                 ; Blittersize errechnen
  lsl.w #6,d6                    ; D5 = breite in Words
  and.w #$3f,d5                  ; D6 = höhe in Pixel
  add.w d6,d5                    ; D5 ist jetzt Blittersize
  move.b 5(a0),d1                ; Anzahl Planes nach D1
  and.w #$00ff,d1                ; Hi-Byte löschen
  sub.w #1,d1                    ; minus 1, wegen DBra
  add.l #8,a0                    ; erster Map-Pointer
gbm_loop:
  move.l (a0)+,$dff050           ; Anfangsadresse  von  Quelle
                                   A
  move.l a2,$dff054              ; Anfangsadresse von Ziel  D
  move.l a2,$dff04c              ; Anfangsadresse  von  Quelle
                                   B = Ziel D
  move.w d5,$dff058              ; BLTSIZE  und  Blitteroperati-
                                   on starten
gbm_wait:
  btst #14,$dff002               ; Blitter fertig?
  bne gbm_wait
  dbra d1,gbm_loop
  clr.l d0                       ; No Errors
  rts
;--- Ende des Listings der Routine GetBackMask () ---
```

Diese Routine reserviert Speicher, der natürlich z.B. bei Beendigung des Programmes oder bei Benutzung eines neuen Bildes an das System zurückgegeben werden sollte. Für diesen Zweck stelle ich ihnen im folgenden die Routine "FreeBackMask ()" vor. Ihr muß in A1 nur die Adresse der ObjektArgs-Struktur (32 Bytes Puffer) übergeben werden, den Rest erledigt sie von selbst.

Kurzbeschreibung der Funktion "FreeBackMask ()":

Routine: FreeBackMask (A1) (ObjektArgs)
Parameter: A1 = Adresse der ObjektArgs-Struktur (32 Bytes Puffer) in der die Schattenmaske für den Hintergrund steht.
Erklärung: Gibt den ShadowMask-Puffer, der mit der Routine "GetBackMask ()" für die Hintergrundmaske reserviert worden ist, wieder an das System zurück.

Listing der Funktion "FreeBackMask ()":

```
;--- ShadowMask-Puffer vom Hintergrund wieder freigeben ---
;--- A1 = Objektargs (32 Bytes Puffer)                   ---
FreeBackMask:
  move.w 12(a1),d0          ; WordWidth nach D0
  lsl.w #1,d0               ; mal 2 = Bytes
  mulu 10(a1),d0            ; mal Höhe = MapSize
  move.l 28(a1),a2          ; CollMask-Zeiger
  clr.l 28(a1)
  move.l a2,a1
  cmp.l #0,a1               ; vorhanden ?
  beq fbm_end
  move.l 4,a6
  jsr -210(a6)              ; FreeMem
fbm_end:
  rts
;--- Ende des Listing der Funktion FreeBackMask () ---
```

5.4 Der RastPort

Da wir auf unserem selbstprogrammierten Bildschirm bzw. BitMap auch Text darstellen wollen, benutzen wir dazu die Systemroutinen der Graphics.library. Der Aufwand wäre viel zu groß, um für diesen Zweck eigene Routinen über die Hardware zu programmieren.

Alle Textroutinen der Graphics.library benötigen als Parameter meist immer einen RastPort. Dieser RastPort wird benötigt, damit das System weiß, wo und wie, in welcher Form der Text usw. dargestellt werden soll. Dieser RastPort besitzt eine Länge von 100 Bytes und muß mit der entsprechenden BitMap-Struktur verbunden werden, in dem der Text dargestellt werden soll.

Wir wollen hier nicht ausführlich auf die RastPort-Struktur eingehen, da es für die Textausgabe nicht unbedingt erforderlich ist. Doch der Vollständigkeit halber folgt eine tabellarische Kurzbeschreibung der RastPort-Struktur:

'RastPort'-Struktur (Länge = 100 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
000	Long	Layer	Zeiger auf 'Layer'-Struktur
004	Long	BitMap	Zeiger auf 'BitMap'-Struktur
008	Long	AreaPtrn	Zeiger auf das AreaFill-Muster
012	Long	TmpRas	Zeiger auf 'TmpRas'-Struktur
016	Long	AreaInfo	Zeiger auf 'AreaInfo'-Struktur
020	Long	GelsInfo	Zeiger auf 'GelsInfo'-Struktur
024	Byte	Mask	Schreibmaske (Bitmapmaske)
025	Byte	FgPen	Farbregisternummer für Vordergrundfarbe
026	Byte	BgPen	Farbregisternummer der Hintergrundfarbe
027	Byte	AOlPen	Farbregisternummer der AreaFill-Outlinefarbe

028	Byte	DrawMode	ZeichenModi (0=Jam1, 1=Jam2, 2=Complement,3=Invers) für Routine 'SetDrMd ()'
029	Byte	AreaPtSz	Höhe des AreaFill-Musters (2^n Words))
030	Byte	LinePatCNT	unbenutzt
031	Byte	Dummy	Line Draw Pattern Preshift
032	Word	Flags	verschiedene Kontrollbits (FirstDot etc.)
034	Word	LinePtrn	16 Bits für Linienmuster
036	Word	CP_X	x-Position des Grafikcursors
038	Word	CP_Y	y-Position des Grafikcursors
040	Byte	Minterms(8)	8 x 1 Byte für Minterms (Funktion unbekannt)
048	Word	PenWidth	Cursorbreite
050	Word	PenHeight	Cursorhöhe
052	Long	TextFont	Zeiger auf 'TextFont'-Struktur
056	Byte	AlgoStyle	Zeichensatzmodus (Style-Flag)
057	Byte	TxFlags	textspezifische Flags
058	Word	TxHeight	Höhe des Zeichensatzes
060	Word	TxWidth	durchschnittliche Zeichenbreite
062	Word	TxBaseLine	Texthöhe ohne Unterlängen
064	Word	TxSpacing	Zeichenabstand
066	Long	RP-User	Zeiger auf evtuelle Userdaten (unwichtig!)
070	Word		7 Words reserviert
084	Long		2 Longs reserviert
092	Byte		8 Bytes reserviert
100	End		Ende der Datenstruktur

Wie aus der RastPort-Struktur zu ersehen ist, muß die Anfangsadresse der BitMap-Struktur ab Offset 4 eingetragen werden. Dies erfüllt zum Beispiel folgender Befehl:

```
move.l #bitmap,rastport+4      ; Bitmap- in RastPort-Struktur einhängen
```

Doch bevor wir Text über unseren RastPort darstellen können, müssen wir diesen mit den korrekten Startparametern füllen. Für diese Aufgabe stellt uns die Graphics.library die Routine "InitRastPort ()" zur Verfügung. Diese Routine braucht lediglich einen Parameter: Wir müssen ihr in A1 die Adresse unserer RastPort-Struktur übergeben.

Routine: InitRastPort (A1) (RastPort)
Offset: -198 = -$c6
Library: graphics.library
Parameter: A1 = Adresse von einem 100 Bytes großen Puffer.
Erklärung: Diese Routine füllt die von uns angelegte RastPort-Struktur mit den nötigsten Startwerten. Diese Routine muß noch mit der BitMap-Struktur verbunden werden. Sonst kann kein Text dargestellt werden.

5.5 Text auf dem Bildschirm ausgeben

Wir wollen hier nur auf die einfache Textausgabe eingehen. Den Text, den wir ausgeben werden, wird immer mit dem aktuellen Zeichensatz und mit der aktuellen Farbe dargestellt. Auf alle Routinen einzugehen, würde den Rahmen des Buches sprengen.

In der Graphicsbibliothek befindet sich eine Funktion, welche den sinnvollen Namen "TEXT" besitzt. Mit ihr ist es möglich, einen Text auf einfache Art und Weise in den aktuellen Parametern, wie Font und Farbe, auszugeben. Um jetzt einen Text ausgeben zu können, übergeben wir in D0 die Anzahl Buchstaben, die der Text lang ist, und in A0 die Anfangsadresse des Textes. Das allein reicht natürlich noch nicht, denn auf welcher Bitmap soll der Text überhaupt ausgegeben werden? Deswegen müssen wir zusätzlich im Adressregister A1 die Adresse der RastPort-Struktur übergeben. Jetzt können wir die Funktion "TEXT ()" aufrufen und der Text wird ausgegeben.

Für einige Anwendungen kann es nützlich sein zu wissen, wie lang ein Text in Pixeln ist, damit zum Beispiel bei dessen Ausgabe keine Grafik zerstört wird. Die Funktion "TextLength ()", ebenfalls aus der Graphicsbibliothek, bietet uns diese Hilfe. Sie besitzt dieselben Parameter wie die Funktion "TEXT ()". Nur bekommen wir nach Aufruf der Funktion im Datenregister D0 die Länge des Textes in Pixeln zurück.

Wenn wir unseren Text auf diese Art darstellen, würde er immer an den aktuellen Koordinaten erscheinen. Diesen Zufallseffekt wollen wir natürlich vermeiden. Um unseren Text an festen Koordinaten erscheinen zu lassen, verwenden wir die Routine "MOVE ()", welche wiederum in der Graphicsbibliothek zu finden ist. Diese Funktion setzt den Grafikcursor an die angegebene Position. In das Adressregister A1 schreiben wir wiederum die Adresse der RastPort-Struktur und in den Datenregistern D0 und D1 die X- und Y-Position. Bei der vertikalen Y-Position ist zu beachten, daß die Nullposition des Textes mindestens die Höhe des Fonts beträgt. Würde man die Y-Position auf Null setzen, wäre der Text auf dem Bildschirm nicht sichtbar. Besitzt der Font zum Beispiel eine Höhe von acht Pixeln (Topaz-Font), wäre der kleinste Y-Wert, bei dem der Text voll erscheinen würde, auch acht.

Der Übersicht halber hier noch einmal alle drei Funktionen in Kurzform auf einen Blick:

Routine: Text (String,RastPort,Count) (A0,A1,D0)
Offset: -60 = -$3c
Library: graphics.library
Parameter: A0 = Anfangsadresse des Textes
A1 = Anfangsadresse der RastPort-Struktur
D0 = Anzahl der Buchstaben, die ausgegeben werden sollen
Erklärung: Diese Routine gibt einen Text im angegebenen RastPort aus.

Routine: TextLength (String,RastPort,Count) (A0,A1,D0)
Offset: -54 = -$36
Library: graphics.library
Parameter: siehe Funktion Text ().
Rückgabe: D0 = Länge des Textes in Pixeln.
Erklärung: Diese Routine gibt für den angegebenen Text die Textlänge in Pixeln im Datenregister D0 zurück.

Routine: Move (RastPort,X,Y) (A1,D0,D1)
Offset: -240 = -$f0
Library: graphics.library
Parameter: A1 = Anfangsadresse der RastPort-Struktur
D0 = X-Position (horizontale) des Textes
D1 = Y-Position (vertikale) des Textes
Erklärung: Setzt den Grafikcursor auf die angegebenen Koordinaten. Danach kann dann zum Beispiel mit der Funktion "Text ()" ab diesen Positionen Text ausgegeben werden.

Die Funktion Text () hat leider einen Nachteil: die Anzahl Buchstaben vom Text muß bekannt sein. Viel leichter wäre es, wenn der Text mit einem Nullbyte enden könnte und somit automatisch das Textende bekannt wäre. Doch wozu haben wir denn einen Computer? Lassen wir ihn doch einfach die Textlänge berechnen. Für diesen Zweck habe ich die einfache Text-Routine ein wenig erweitert, so daß sie einen gewünschten Text, der mit einem Nullbyte endet, an bestimmten Koordinaten erscheinen lassen können.

Das folgende Listing erfüllt diesen Zweck. Welche Parameter übergeben werden müssen, steht am Anfang des Listings. Natürlich können sie diese Routine beliebig erweitern, wie zum Beispiel die Erkennung und entsprechender Verarbeitung von Enterzeichen (ASCII-Wert = 13).

```
;--- Gibt einen Text auf dem Bildschirm aus,      ---
;--- der mit einem Nullbyte endet                 ---
;--- Parameter: A0 = Text, A1 = RastPort,         ---
;--- A6 = Graphics-Basis                          ---
;--- D0 = X-Pos., D1 = Y-Position                 ---

PrintText:
  clr.w d2                      ; Zähler für Anzahl Zeichen
  move.l a0,a2
pt_loop:
  tst.b (a2)+
  beq pt_loop_end
  add.w #1,d2
  cmp.w #100,d2                 ; Max. 80 Zeichen
  bne pt_loop
pt_loop_end:
  tst.w d2                      ; Kein Text ?
  beq pt_end
  movem.l a0-a1/a6/d2,-(sp)     ; Parameter retten
  jsr -240(a6)                  ; MOVE () - Position setzen
  movem.l (sp)+,a0-a1/a6/d0     ; Parameter holen
  jsr -60(a6)                   ; TEXT () - Text printen
pt_end:
  rts
;--- Listingende ---
```

Stellen sie sich mal vor, sie haben den ganzen Bildschirm voll mit Text und wollen auf Tastendruck eine weitere Textseite darstellen. Dazu muß die alte gelöscht werden. Die Grahicsbibliothek stellt uns dafür zwar eine Funktion zur Verfügung, doch ist diese nicht gerade sehr leistungsfähig. Viel besser geht es mit einer selbstgeschriebenen Routine. Solch eine Routine folgt weiter unten, welche rein über die Hardware läuft und außerdem zusätzliche Möglichkeiten beinhaltet. Diese Routine habe ich den Namen FillBitMap() gegeben. Ihr kann im Datenregister D0 ein 16-Bit großes Muster übergeben werden, mit dem die BitMap gefüllt werden soll. Übergibt man den Wert Null, so wird der Bildschirm gelöscht.

Routine: FillBitMap (D0,A1) (Muster,BitMap)
Parameter: D0 = Füllmuster (16 Bit bzw. 1 Word groß), mit dem die BitMap gefüllt werden soll.
A1 = Adresse der BitMap-Struktur
Erklärung: Diese Routine füllt eine Bitmap auf die A1 zeigt, mit einem wordgroßen Muster, welches in D0 übergeben wird. Die BitMap darf maximal die Ausmaße 1024 x 1024 haben.

```
;--- Routine zum Füllen der Bitmap mit dem Blitter      ---
;--- D0 = FillMuster, A1 = BitMap                       ---
FillBitMap:
  move.l #$dff000,a6
  move.w d0,$74(a6)              ; Fillmuster
  move.l #$01f00000,$40(a6)      ; USE A und D1,Minterm:A=D
  move.l #-1,$44(a6)             ; First/Last Mask (alle Bits
                                   übernehmen)
  clr.l $64(a6)                  ; Modulowert von Quelle A
  move.w (a1),d5                 ; Breite in Bytes
  lsr.w #1,d5                    ; jetzt in Words
  move.w 2(a1),d6                ; Höhe in Pixel
  and.w #$3ff,d6                 ; Blittersize errechnen
  lsl.w #6,d6                    ; D5 = breite in Words
  and.w #$3f,d5                  ; D6 = höhe in Pixel
  add.w d6,d5                    ; D5 ist jetzt Blittersize
  clr.w d0
  move.b 5(a1),d0                ; Tiefe
  subq #1,d0                     ; minus 1
  add.l #8,a1
fbmap_loop:
  move.l (a1)+,$54(a6)           ; Anfangsadresse von Ziel D
  move.w d5,$58(a6)              ; BLTSIZE und Blitteroperati-
                                   on starten
fbmap_wait:
  btst #14,$2(a6)                ; Warten bis Blit fertig
  bne fbmap_wait
  dbra d0,fbmap_loop
  rts                            ;--- Listingende ---
```

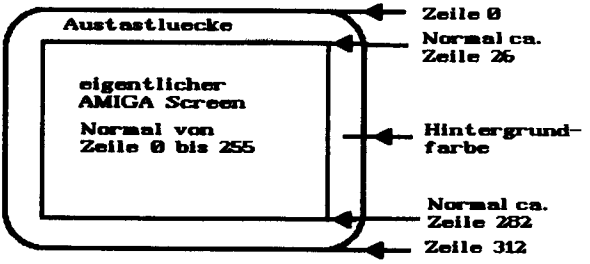
5.6 Der Copper

Der Copper ist ein Co-Prozessor, der neben dem 68000er Prozessor zusätzlich programmiert werden kann. Der Copper besitzt ein eigenes Programm, welches er abfährt. Dabei richtet er sich genau nach dem Bildschirmaufbau, was zur Folge hat, daß er nur 50 mal in der Sekunde sein Programm abfährt. Außerdem kennt er auch nur drei Befehle. Diese sind der Move-, Wait- und Skip-Befehl, wobei letzterer kaum Anwendung findet.

Der Copper wird eigentlich nur für die Darstellung von Bildern gebraucht, damit die Hardwareregister, in denen die Adressen der Bitmaps ständig hochgezählt werden, am Ende auch wieder zum richtigen Zeitpunkt auf den Anfangswert gesetzt werden. Denn sonst würde ständig ein anderes Bild erscheinen. Dieser Flackereffeckt ist ihnen bestimmt schon mal bei einigen Abstürzen (Guru...) aufgefallen.

Mit dem Copper kann man aber auch durch geschickte Programmierung die 32 Hardwarefarbregister mehrmals benutzten, verschiedene Auflösungen auf einmal, Interrupt, Farbspielerein usw. erzeugen. Erst durch den Copper ist das System in der Lage, mehrere Screens zu erzeugen und diese überlappen zu lassen.

Bildschirmzeileneinteilung fuer den Copper



5.7 Die Copperliste

Wie schon erwähnt wurde, kennt der Co-Prozessor nur drei Befehle. Diese sind der Move-, Wait- und Skip-Befehl. Das Programm für den Co-Prozessor nennt man Copperlist. In dieser Liste liegen die Befehle direkt hintereinander, wobei jeder Befehl aus zwei 16 Bit-Werten besteht.

Der MOVE-Befehl:

Format: dc.w Customregister,Wert

Mit dem Move-Befehl kann man einen 16-Bit-Wert in ein Hardwareregister übertragen. Dabei handelt es sich um ein Customchip-Register. Zur Verfügung stehen einem die Hardwareregister von $DFF080 bis $DFF1BE. Setzt man zusätzlich das COPCON-Register ($DFF02E) auf 1, ist man in der Lage, auf die Register von $DFF040 bis $DFF07E zuzugreifen. Damit besteht die Möglichkeit, über eine Copperliste den Blitter zu programmieren.

Den MOVE-Befehl kann man daran erkennen, daß im ersten Befehlsword (16-Bit-Wert) das Bit 0 = 0 ist. Außerdem enthält dieses Befehlsword die Registeradresse, die mit dem zweiten Befehlsword geladen werden soll. Bei den Registeradressen wird die Basisadresse der Customchips ($DFF000) weggelassen. Es wird nur der Offset ins erste Befehlsword eingetragen.

Beispiel: dc.w $180,1000

Bei diesem Beispiel wird in das Hintergrundfarbregister $DFF180 der Farbwert 1000 eingetragen. Um das Bit 0 im ersten Befehlsword braucht man sich nicht kümmern, da die Customchips alle auf geraden Adressen liegen und somit Bit 0 immer Null ist.

Der Wait-Befehl:

Format: dc.w Rasterzeile,Maske

Mit dem Wait-Befehl ist man in der Lage, auf eine beliebige Rasterstrahlposition zu warten. Wird ein Wait-Befehl in der Copperliste erreicht, dann vergleicht der Copper den angegebenen Wert mit der aktuellen Rasterposition. Ist der Rasterstrahl schon weiter, so wird in der Copperliste der nächste Befehl bearbeitet. Ansonsten wird so lange mit der Bearbeitung gewartet, bis die Rasterposition erreicht wurde.

Das zweite Befehlsword enthält die dazugehörige Maske, denn nur wenn ein Maskenbit und das dazugehörige Positionsbit gesetzt sind, wird es zum Vergleich mit der Rasterstrahlposition herangezogen. Die Bitbelegung der beiden Befehlsworte ist folgende:

Befehlsword 1:

```
Bit: 15  14  13  12  11  10  09  08
     VP7 VP6 VP5 VP4 VP3 VP2 VP1 VP0
Bit: 07  06  05  04  03  02  01  00
     HP8 HP7 HP6 HP5 HP4 HP3 HP2 1
```

Befehlsword 2:

```
Bit: 15  14  13  12  11  10  09  08
     BFD VM6 VM5 VM4 VM3 VM2 VM1 VM0
Bit: 07  06  05  04  03  02  01  00
     HM8 HM7 HM6 HM5 HM4 HM3 HM2 0
```

VP = Vertikale Rasterstrahlposition
HP = Horizontale Rasterstrahlposition
VM = Vertikale Maskenbits
HM = Horizontale Maskenbits

Den Wait-Befehl erkennt man daran, daß im ersten Befehlsword das Bit 0 auf 1 und im zweiten Befehlsword auf 0 ist.

Wie schon aus dem ersten Befehlsword ersichtlich wird, stehen einem für die vertikale Rasterstrahlposition nur acht Bits (VP7 bis VP0) zur Verfügung, wodurch nur ein maximaler Wertebereich von 0 bis 255 erreicht werden kann. Hingegen gibt es aber 313 mögliche Zeilen. Will man jetzt auf eine Rasterzeile ab 256 warten, so muß man erst mit einem Wait-Befehl auf die Zeile 255 warten und danach unter Vernachlässigung des 9. Bits auf die gewünschte Zeile.

Für die horizontale Rasterposition existieren sogar nur 7 Bits, was 112 mögliche Positionen zuläßt. Damit läßt sich die horizontale Koordinate nur in Viererschritten angeben.

Damit der Copper weiß, wann die Copperliste zu Ende ist, wird am Schluß jeder Copperliste mit einem Wait-Befehl auf eine unmögliche Rasterposition gewartet. Der folgende Wait-Befehl erfüllt diese Bedingung:

```
dc.w $FFFF,$FFFE
```

Nach diesem Wait-Bedingung fährt der Copper sein Programm wieder von vorne ab.

Der Skip-Befehl:

Format: dc.w Rasterzeile,Maske

Der SKIP-Befehl ist genauso aufgebaut wie der WAIT-Befehl, nur daß das Bit 0 in beiden Befehlswörtern auf 1 ist. Der Skip-Befehl prüft, ob die angegebene Rasterposition kleiner der tatsächlichen ist. Ist dieses der Fall, so überspringt der Copper den nächsten Befehl in der Copperliste, andernfalls fährt er mit dieser fort. Damit lassen sich bedingte Verzweigungen simulieren. Dieser Befehl wird aber so gut wie nie benutzt.

Initialisieren und Starten einer Copperliste

Hat man jetzt seine Copperliste ordnungsgemäß aufgebaut, so wird die Anfangsadresse dieser in die Hardwareregister $DFF080 und $DFF082 geschrieben. Das kann mit einem MOVE.L -Befehl erledigt werden. Damit jetzt auch unsere Copperliste gestartet wird, schreiben wir einen X-beliebigen Wert in das Hardwareregister $DFF088.

Hier ein Beispiel:

```
  move.l #copperlist,$dff080      ; Adresse der Copperliste
  clr.w $dff088                   ; Copperliste starten
  rts
Copperliste:
  dc.w $180,0                     ; MOVE
  dc.w $ffff,$fffe                ; Ende der Copperlist
```

Um jetzt auch wieder auf die alte Copperliste zurückschalten zu können, brauchen wir deren Adresse. Die finden wir in der Graphicsbasis ab Offset 38.

Beipiel zum Zurückschalten auf die alte Copperliste:

```
Old_Copperlist:
  move.l gfxbase,a0               ; Basisadresse der graphics.library
  move.l 38(a0),$dff080           ; Adresse der alten Copperliste
  clr.w $dff088                   ; und starten
  rts
```

Natürlich darf nicht vergessen werden, den Copper-DMA anzuschalten. Aber dies können wir hier vernachlässigen, da dieser immer angeschaltet ist. Nur wenn wir ihn ausschalten, sollten wir dieses beachten, wenn wir eine Copperliste verwenden wollen.

5.8 Copperroutinen

Bevor überhaupt ein Bild über den Rasterstrahl erscheinen kann, muß bekannt sein, wo er anfangen und wo er mit seiner Arbeit enden soll, außerdem wie groß das eigentliche "Fenster" ist, in dem die Bitmap über den Rasterstrahl dargestellt wird. Ihnen ist bestimmt das Preferencesprogramm bekannt, mit dem eine Verschiebung mit der Maus, des Screens innerhalb eines bestimmten Raumes möglich ist. Nur die eigentliche Fenstergröße ist nicht änderbar.

Für das Setzen des Anfangs- und Endpunktes, nach dem der Rasterstrahl dann seinen Verlauf nimmt, sind die Customregister DIWSTRT ($DFF08E) und DIWSTOP ($DFF090) verantwortlich. Das DIWSTRT-Register ist für den Anfangspunkt und das DIWSTOP-Register für den Endpunkt zuständig. Für die Anfangs- und Endpunkte zur Darstellung des eigentlichen Bitmap-"Fensters", sind die Customregister DDFSTRT ($DFF092) und DDFSTOP ($DFF094) vorhanden.

Die Beschreibung dieser Register ist sehr kompliziert und hier wird deswegen auch nicht weiter darauf eingegangen. Damit wir aber diese Register korrekt initialisieren können, stelle ich Ihnen hier Beispielwerte vor, mit denen die Register meistens geladen werden.

DIWSTRT = $3081
DIWSTOP = $30C1
DDFSTRT = $0038
DDFSTOP = $00D0

Beispiellisting:

```
move.w #$3081,$dff08e
move.w #$30c1,$dff090
move.w #$0038,$dff092
move.w #$00d0,$dff094
```

oder als Copperliste:

```
dc.w $8e,$3081          ; MOVE
dc.w $90,$30c1          ; MOVE
dc.w $92,$0038          ; MOVE
dc.w $94,$00d0          ; MOVE
```

Aber das alleine reicht, wie kann es auch anders sein, nicht zur Darstellung eines Bildes. Denn woher soll der Amiga wissen, ab welcher Adresse die Bitmap im Speicher liegt, welcher Grafikmodus vorliegt usw.? Die Customregister, die dafür zuständig sind, müssen vor der Darstellung mit den dazugehörigen Werten geladen werden. Da wir uns aber nicht mit den komplizierten Hardwareregistern herumschlagen wollen, stelle ich an dieser Stelle zwei Routinen vor, die diese Aufgaben kinderleicht bewältigen.

Die erste Routine, die sich InitColor () nennt, initialisiert die Farbregister des Amiga mit den gewünschten Farben. Dazu muß in der Copperliste ein Puffer von 128 Bytes vorhanden sein, der nach Aufruf der Routine mit den dazugehörigen MOVE-Befehlen für die Farbregister gefüllt wird. Diese Routine sollte vor Inbetriebnahme der Copperliste einmal aufgerufen werden, damit die richtigen MOVE-Befehle schon einmal in der Copperliste vorhanden sind. Danach kann diese Routine so oft aufgerufen werden, wie man möchte bzw. so oft man die Farben ändern möchte.

Die zweite Routine, die sich InitCopperMap () nennt, initialisiert die wichtigsten Customregister, die für die Darstellung eines Screens erforderlich sind. Diese Funktion benötigt einen 60 Bytes großen Puffer, in dem nach Aufruf die dazugehörigen MOVE-Befehle abgelegt werden können. Auch diese Routine sollte vor dem Starten der Copperliste, aus denselben Gründen, wie bei der Funktion InitColor (), einmal aufgerufen werden.

Routine: InitColor (A0,A1) (ColorTable,Copperpuffer)
Parameter: A0 = Zeiger auf einen Puffer von 32 Words in dem die Farbwerte für die 32 Farbregister stehen.
A1 = Zeiger auf einen 128 Bytes großen Puffer, innerhalb der Copperliste, in den die 32 Farben kopiert werden.
Erklärung: Kopiert die 32 Farben, auf die ColorTable zeigt, in den 128 Bytes großen Copperpuffer und erzeugt auch die dazugehörigen MOVE-Befehle.
Achtung! - Diese Routine beim ersten Mal vor dem Aktivieren der Copperliste aufrufen, damit die Farbregister ($180 bis ...) schon einmal in der Copperliste enthalten sind. Danach kann man diese Routine jederzeit wieder aufrufen und damit die Farben ändern.

Routine: InitCopperMap(A0,A1,D0) (BitMap,Copperpuffer,Modus)
Parameter: A0 = Zeiger auf BitMap-Struktur
A1 = Zeiger auf einen 60 Bytes großen Puffer, innerhalb der Copperliste, in dem die Register kopiert werden.
D0 = Grafik-Modus:
= $8000 für Hires
= $800 für HAM (Hold and Modify)
= $400 für DualPlayField
= $4 für Interlace
= $0 für Normal
Erklärung: Setzt die wichtigsten Startwerte zum aktivieren der Grafik. Es werden die Customregister BPL1MOD, BPL2MOD, BPLCON0 und die BitMap-Pointer mit den entsprechenden Werten gefüllt. Diese Register werden dann in den 60 Bytes großen Puffer, innerhalb der Copperliste, kopiert.
Achtung! - siehe Achtung! bei InitColor ().
Achtung! - Wir verwenden in diesem Buch nur den Normalmodus, weil die anderen einer weiteren Erklärung bedürfen. Sie sind für die Spieleprogrammierung nicht besonders geeignet.

Hier jetzt noch die beiden Assemblerlistings zu den Routinen:

```
;--- ColorMap in CopperListe übertragen          ---
;--- A0 = ColorTable ; A1 = CopperPuffer         ---
InitColor:
  move.w #$180,d1              ; erstes Farbregister
  move.w #31,d0                ; Anzahl Farben
ic_loop:
  move.w d1,(a1)+              ; Farbregister in CopperList
  move.w (a0)+,(a1)+           ; dann der Farbwert in Cop-
                                 perList
  add.w #2,d1                  ; nächstes Farbregister
  dbra d0,ic_loop
  rts
;--- Listingende ---

;--- BitMap-Pointer in CopperList übertragen     ---
;--- A0 = BitMap ; A1 = CopperPuffer ; D0 = Modus ---
InitCopperMap:
  move.w (a0),d1               ; Bytes pro Zeile nach D1
  sub.w #40,d1                 ; minus 40 Bytes (320 Pixel)
  cmp.w #$8000,d0              ; Modus = Hires ?
  bne icm_1
  sub.w #40,d1                 ; nochmal minus 40 Bytes
                                 (insgesammt -640 Pixel)
icm_1:
  cmp.w #$04,d0                ; Modus = Interlace ?
  bne icm_2
  sub.w #40,d1                 ; auch minus 40 Bytes (ins-
                                 gesammt -640 Pixel)
icm_2:
  move.w #$0108,(a1)+          ; BPL1MOD
  move.w d1,(a1)+              ; Modulo-Wert ungerade Pla-
                                 nes
  move.w #$010a,(a1)+          ; BPL2MOD
  move.w d1,(a1)+              ; Modulo-Wert gerade Planes
  move.b 5(a0),d1              ; Depth nach D1
  lsl.w #8,d1                  ; Korrekten
```

```
  lsl.w #5,d1                ; Depth-Wert
  lsr.w #1,d1                ; ermitteln
  add.w #$0200,d1            ; Color setzen
  add.w d0,d1                ; Modus setzen
  move.w #$0100,(a1)+        ; BPLCON0
  move.w d1,(a1)+            ; setzen
  moveq #5,d0                ; max. Tiefe minus 1
  move.w #$00e0,d1           ; Hi-Word vom ersten Map-
                               Pointer
  add.l #8,a0                ; Erster Pointer-Zeiger
icm_loop:
  move.w d1,(a1)+            ; HI-Word vom Pointer
  move.w (a0)+,(a1)+         ; setzen
  add.w #2,d1                ; Lo-Word ermitteln
  move.w d1,(a1)+            ; und
  move.w (a0)+,(a1)+         ; setzen
  add.w #2,d1                ; Hi-Word ermitteln
  dbra d0,icm_loop
  rts
;--- Listingende ---
```

Damit das ganze auch besser verstanden wird, folgt nun noch ein Beispiellisting, das eine Copperliste korrekt initialisiert.

```
;--- Startet eine Copperlist und auf rechte Maustaste      ---
;--- wird Programm beendet und alte Copperlist             ---
;--- eingeschaltet                                         ---
;
start_copperlist:
  lea gfxname,a1             ; Libraryname
  clr.l d0                   ; Version = Null
  move.l 4,a6                ; Execbase
  jsr -552(a6)               ; OpenLibrary ()
  move.l d0,gfxbase          ; graphics.basis
;
  lea colortable,a0          ; Zeiger auf Farbtabelle
  lea colorpuffer,a1         ; Zeiger auf Copperpuffer
  bsr initcolor              ; Farben initialisieren
```

```
  lea bitmap,a0                    ; Zeiger auf BitMap
  lea copperpuffer,a1              ; Zeiger auf Copperpuffer
  clr.w d0                         ; Modus = Normal
  bsr initcoppermap                ; Wichtigsten Werte setzen
  move.l #copperlist,$dff080       ; Neue Copperlistadresse
  clr.w $dff088                    ; und Copperliste starten
wait:
  btst #10,$dff016                 ; rechte Maustaste gedrückt ?
  bne wait
  move.l gfxbase,a0                ; Graphics.basis
  move.l 38(a0),$dff080            ; alte Copperlistadresse
  clr.w $dff088                    ; und starten
;
  move.l gfxbase,a1                ; graphics.basis
  move.l 4,a6                      ; Execbasis
  jsr -414(a6)                     ; CloseLibrary ()
  rts
;
;--- Parameter ---
gfxbase: dc.l 0
gfxname: dc.b "graphics.library",0
  even
bitmap: blk.b 40,0
colortable: blk.w 32,1000

copperlist:
  dc.w $8e,$3081
  dc.w $90,$30c1
  dc.w $92,$38
  dc.w $94,$d0
colorpuffer: blk.b 128,0
copperpuffer: blk.b 60,0
  dc.l $fffffffe                   ; Ende der Copperlist
;
;--- Listingende ---
```

5.9 Einen Screen (Bild) programmieren

In diesem Abschnitt werden wir endlich dazu übergehen, mit unserem bisherigen Wissen, einen Screen über die Hardware zu programmieren. Als erstes folgt das Assemblerlisting. Darauf folgt die ausführliche Beschreibung. An dem Listing werden sie schon merken, wie kinderleicht es ist, einen Screen zu programmieren.

Das Listing erstellt einen Screen und läßt darauf einen Text erscheinen. Auf Druck der rechten Maustaste wird auf die alte Copperliste zurückgeschaltet und damit das Programm beendet.

```
;--- Erzeugt einen Screen über die Hardware                    ---
;--- Printet einen Text  und auf rechte Maustaste erscheint    ---
;--- alte Copperliste und Programm wird beendet                ---
;--- Programmname = Display                                    ---
;
  move.l 4,a6                   ; ExecBasis
  lea gfxname,a1                ; Libraryname
  clr.l d0                      ; Version = 0
  jsr -552(a6)                  ; OpenLibrary ()
  move.l d0,gfxbase             ; graphics.basis
;
  lea rastport,a1               ; RastPort-Struktur
  move.l gfxbase,a6             ; graphics.basis
  jsr -198(a6)                  ; InitRastPort ()
;
  lea bitmap,a0                 ; BitMap-Struktur
  move.l #3,d0                  ; Depth = 3
  move.l #320,d1                ; 320 breit
  move.l #256,d2                ; 256 hoch
  move.l #1,d3                  ; Speicher reservieren
  bsr initbitmap                ; BitMap init.
;
  lea bitmap,a0                 ; BitMap-Struktur
  lea rastport,a1               ; RastPort-Struktur
```

```
  move.l a0,4(a1)                ; BitMap in RastPort einhän-
                                   gen
;
  lea colortable,a0              ; Farbtabelle
  lea colorpuffer,a1             ; Copperpuffer
  bsr initcolor                  ; Farbregister init.
;
  lea bitmap,a0                  ; BitMap-Struktur
  lea copperpuffer,a1            ; Copperpuffer
  clr.l d0                       ; Modus = Normal
  bsr initcoppermap              ; Customregister init.
;
  move.l #copperlist,$dff080     ; Copperlistadresse
  clr.w $dff088                  ; Copperliste starten
;
  move.w #$20,$dff096            ; Sprite-DMA aus
  move.l #spritedatas,$dff120    ; Sprite 0 (Mauszeiger) aus
;
  lea demotext,a0                ; Textadresse
  lea rastport,a1                ; RastPort-Struktur
  move.l gfxbase,a6              ; graphics.basis
  clr.l d0                       ; X-Position
  move.l #20,d1                  ; Y-Position
  bsr printtext                  ; Text printen
;
;--- Maustaste abfragen ---
main:
  btst #10,$dff016               ; Rechte Maustaste ?
  bne main
;
;--- Programm beenden ---
  move.l gfxbase,a0              ; graphics.basis
  move.l 38(a0),$dff080          ; alte Copperlistadresse
  clr.w $dff088                  ; und starten
;
  move.w #$8220,$dff096          ; Sprite-DMA an
;
  lea bitmap,a0                  ; BitMap-Struktur
```

```
  bsr clearbitmap                ; BitMap freigeben
;
  move.l gfxbase,a1              ; graphics.basis
  move.l 4,a6                    ; exec.basis
  jsr -414(a6)                   ; CloseLibrary ()
;
  rts                            ; Programmende

;--- Gibt einen Text auf dem Bildschirm aus,
;--- der mit einem Nullbyte endet
;--- Parameter: A0 = Text, A1 = RastPort, A6 = Graphics-Basis
;--- D0 = X-Pos., D1 = Y-Position
PrintText:
  clr.w d2                       ; Zähler für Anzahl Zeichen
  move.l a0,a2
pt_loop:
  tst.b (a2)+
  beq pt_loop_end
  add.w #1,d2
  cmp.w #100,d2                  ; Max. 80 Zeichen
  bne pt_loop
pt_loop_end:
  tst.w d2                       ; Kein Text ?
  beq pt_end
  movem.l a0-a1/a6/d2,-(sp)      ; Parameter retten
  jsr -240(a6)                   ; MOVE () - Position setzen
  movem.l (sp)+,a0-a1/a6/d0      ; Parameter holen
  jsr -60(a6)                    ; TEXT () - Text printen
pt_end:
  rts

;--- BitMap-Pointer in CopperList übertragen
;--- A0 = BitMap ; A1 = CopperPuffer ; D0 = Modus
InitCopperMap:
  move.w (a0),d1                 ; Bytes pro Zeile nach D1
  sub.w #40,d1                   ; minus 40 Bytes (320 Pixel)
  cmp.w #$8000,d0                ; Modus = Hires ?
  bne icm_1
```

```
  sub.w #40,d1                ; nochmal minus 40 Bytes
                                (insgesammt -640 Pixel)
icm_1:
  cmp.w #$04,d0               ; Modus = Interlace ?
  bne icm_2
  sub.w #40,d1                ; auch minus 40 Bytes (ins-
                                gesammt -640 Pixel)
icm_2:
  move.w #$0108,(a1)+         ; BPL1MOD
  move.w d1,(a1)+             ; Modulo-Wert ungerade Pla-
                                nes
  move.w #$010a,(a1)+         ; BPL2MOD
  move.w d1,(a1)+             ; Modulo-Wert gerade Planes
  move.b 5(a0),d1             ; Depth nach D1
  lsl.w #8,d1                 ; Korrekten
  lsl.w #5,d1                 ; Depth-Wert
  lsr.w #1,d1                 ; ermitteln
  add.w #$0200,d1             ; Color setzen
  add.w d0,d1                 ; Modus setzen
  move.w #$0100,(a1)+         ; BPLCON0
  move.w d1,(a1)+             ; setzen
  moveq #5,d0                 ; max. Tiefe minus 1
  move.w #$00e0,d1            ; Hi-Word vom ersten Map-
                                Pointer
  add.l #8,a0                 ; Erster Pointer-Zeiger
icm_loop:
  move.w d1,(a1)+             ; HI-Word vom Pointer
  move.w (a0)+,(a1)+          ; setzen
  add.w #2,d1                 ; Lo-Word ermitteln
  move.w d1,(a1)+             ; und
  move.w (a0)+,(a1)+          ; setzen
  add.w #2,d1                 ; Hi-Word ermitteln
  dbra d0,icm_loop
  rts

;---- ColorMap in CopperListe übertragen
;--- A0 = ColorTable ; A1 = CopperPuffer
InitColor:
```

```
  move.w #$180,d1          ; erstes Farbregister
  move.w #31,d0            ; Anzahl Farben
ic_loop:
  move.w d1,(a1)+          ; Farbregister in CopperList
  move.w (a0)+,(a1)+       ; dann der Farbwert in Cop-
                             perList
  add.w #2,d1              ; nächstes Farbregister
  dbra d0,ic_loop
  rts

;--- BitMap-Struktur initialisieren und Speicher
;--- qfür Maps reserverieren
;--- D0 = Depth, D1 = Width, D2 = Height, D3 = Memory
;--- A0 = BitMap
InitBitMap:
  move.w d1,d4             ; Breite nach D4
  and.w #15,d4             ; Rest ermitteln
  tst.w d4                 ; Rest vorhanden ?
  beq ibm_1                ; Wenn nicht, dann weiter -
                             sonst Error
  move.l #-1,d0            ; Error: Breite nicht durch 16
                             teilbar
  rts
ibm_1:
  lsr.w #4,d1              ; Breite durch 16 teilen
  lsl.w #1,d1              ; mal 2 = Anzahl Bytes einer
                             Zeile
  move.w d1,(a0)           ; und in BitMap-Struktur
                             speichern
  move.w d2,2(a0)          ; Höhe in BitMap-Struktur
                             speichern
  move.w d0,4(a0)          ; Anzahl Planes speichern
  tst.w d3                 ; Allocate Speicher für Bit-
                             Maps?
  bne ibm_2                ; Wenn nicht, dann Ende,
                             sonst weiter
  clr.l d0                 ; No Errors
  rts                      ; Ende
```

```
ibm_2:
  move.w d0,d4                ; Depth nach D4
  sub.w #1,d4                 ; minus 1
  lea 8(a0),a1                ; BitMap nach a1
ibm_clear_loop:               ; Pointer-Zeiger
  clr.l (a1)+                 ; löschen
  dbra d4,ibm_clear_loop
  mulu d2,d1                  ; BytePerRow x Höhe = Size
                                von einer Map
  move.w d0,d2                ; D2 = Anzahl Planes
  move.l d1,d0                ; D0 = ByteSize
  move.l #$10002,d1           ; D1 = CHIP + FreeMem
  sub.w #1,d2                 ; Anzahl Planes minus 1, we-
                                gen DBRA
  add.l #8,a0                 ; Anfang BitMap-Zeiger
  move.l a0,a2                ; auch nach A2
ibm_loop:
  move.l #$4,a6
  move.l (a6),a6
  movem.l d0-d2/a0-a2,-(sp)
  jsr -198(a6)                ; AllocMem
  tst.l d0                    ; konnte Speicher reserviert
                                werden ?
  bne ibm_l1                  ; Wenn ja, dann weiter
  movem.l (sp)+,d0-d2/a0-a2
ibm_free_loop:
  move.l (a2),a1              ; Zeiger auf BitMap holen
  cmp.l #0,a1                 ; Null ?
  bne ibm_l2                  ; wenn ja, dann ende
  move.l #-2,d0               ; Error
  rts                         ; Ende
ibm_l2:
  clr.l (a2)+
  movem.l d0/a2,-(sp)
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                ; FreeMem
  movem.l (sp)+,d0/a2
```

```
  bra ibm_free_loop
ibm_l1:
  move.l d0,d5                    ; Zeiger auf BitMap nach D5
  movem.l (sp)+,d0-d2/a0-a2
  move.l d5,(a0)+
  dbra d2,ibm_loop
  clr.l d0
  rts

;--- Speicher der Maps wieder freigeben
;--- in einer BitMap-Struktur
;--- A0 = BitMap
ClearBitMap:
  clr.w d1
  move.b 5(a0),d1                 ; Depth nach D1
  sub.w #1,d1
  move.w (a0),d0                  ; BytePerRow
  mulu 2(a0),d0                   ; mal Höhe = MapSize
  add.l #8,a0
  move.l #$4,a6
  move.l (a6),a6
cbm_loop:
  move.l (a0),a1                  ; Map-Pointer
  cmp.l #0,a1
  beq cbm_l1
  clr.l (a0)+
  movem.l d0-d1/a0/a6,-(sp)
  jsr -210(a6)                    ; FreeMem
  movem.l (sp)+,d0-d1/a0/a6
cbm_l1:
  dbra d1,cbm_loop
  rts

;--- Parameter ---
gfxbase: dc.l 0                   ; graphics.basis
gfxname: dc.b "graphics.library",0  ; Libraryname
  even
bitmap: blk.b 40,0                ; BitMap-Struktur
```

```
rastport: blk.b 100,0                ; RastPort-Struktur
colortable:
  dc.w 0,1000,2000,3000
  blk.w 28,235                       ; 32 Farben
spritedatas: blk.l 3,0
copperlist:                          ; ab hier Copperliste
  dc.w $8e,$3081                     ; DIWSTRT
  dc.w $90,$30c1                     ; DIWSTOP
  dc.w $92,$38                       ; DDFSTRT
  dc.w $94,$d0                       ; DDFSTOP
colorpuffer:
  blk.b 128,0                        ; Farbregister
copperpuffer:
  blk.b 60,0                         ; Screen-Customregister
  dc.l $fffffffe                     ; Copperlistende
demotext:
  dc.b "Dieses ist ein Demo-Screen!",0
  even
  END
;--- Listingende ---
```

Beschreibung

Als erstes wird die Graphicslibrary geöffnet, um die Text-Funktionen aus dieser Library benutzen können. Außerdem müssen wir am Ende des Programms wieder auf die alte Copperliste zurückschalten können. Jetzt initialisieren wir die RastPort- und BitMap-Struktur und verbinden diese miteinander. Werden diese Schritte vergessen, erscheint über die Textroutinen kein Text.

Da wir unseren Screen über eine eigene Copperliste erzeugen, muß diese natürlich auch korrekt initialisiert werden. Dafür sorgen die nächsten zwei Schritte. InitColor () legt die 32 Farbregister und InitCopperMap () die wichtigsten Customregister in die Copperliste ab. Danach wird die Copperliste ordnungsgemäß gestartet.

Wenn die System-Copperliste durch eine neue ersetzt wird, wie in unserem Listing, hat dies einen unschönen Effekt zur Folge. Es erscheint entweder ein dicker oranger Balken oder ein flackernder Streifen. Dieser Balken bzw. Streifen ist der Mauszeiger, welcher nicht mehr richtig über den Sprite-DMA dargestellt werden kann, weil in unserer Copperliste die Spriteregister nicht mehr mit den richtigen Werten versorgt werden. Damit dieser Effekt nicht auftritt, schalten wir einfach den Sprite-DMA-Kanal aus und setzen die Dataadresse von Sprite0 (Mauszeiger) auf einen Puffer von Nullbytes.

Es wäre ja langweilig, wenn wir nur ins Schwarze gucken müßten. Darum printen wir zur Abwechslung noch einen Text auf den Screen. In einer Schleife wird jetzt abgefragt, ob die rechte Maustaste gedrückt wird. Ist dieses der Fall, wird das Programm beendet. Wir schalten auf die alte Copperliste zurück und den Sprite-DMA-Kanal wieder ein. Als letztes wird der Speicher der BitMaps zurückgegeben und die Graphicslibrary wieder geschlossen.

Kapitel 6

Joystickabfrage

Was wäre ein Actionspiel ohne Action. Absolut langweilig, selbst wenn es eine hervoragende Graphik besitzen würde. Nach einiger Zeit wäre auch diese fade. Deswegen sollte man beim Spielen schon ins "schwitzen" kommen. Dies geht natürlich nur über den Joystick. Denn dieser Steuerknüppel ist, neben der Tastatur oder Maus, die einzige Verbindung zum Computer.

Das Problem, das sich uns nun stellt, ist die Programmierung dieser Joystickabfrage. Es gibt mehrere Möglichkeiten, die Ports, welche die Verbindung zum Joystick herstellen, zu programmieren. Einige sind sehr kompliziert, andere funktionieren nicht richtig und so weiter...

Dabei ist es sehr einfach die Ports richtig zu programmieren. Denn für die beiden Ports existiert jeweils ein dazugehöriges Customregister. Das Portregister 1 besitzt die Adresse $DFF00A und das Portregister 2 ist ab der Adresse $DFF00C zu finden.

PORT1 = $DFF00A PORT2 = $DFF00C

Wenn wir den Joystick in eine bestimmte Richtung drücken, wird in das dazugehörige Portregister ein Wert geschrieben. Da man den Joystick in acht verschiedene Richtungen drücken kann, gibt es auch acht verschiedene Werte, passend jeweils zur Richtung. Da die Portregister Leseregister sind, braucht nur der jeweilige Richtungswert mit dem Wert in diesem Register abgefragt und bei gleichem Ergebnis entsprechend verzweigt werden. So einfach ist die Joystickprogrammierung.

Wenn wir diese Methode verwenden, kann es allerdings passieren, daß sich die Werte ab und zu mal ändern und dadurch die Bewegung der Figur zum Beispiel nicht synchron mit der Joystikkrichtung ausfällt. Diesen Fehler können wir leicht umgehen. Denn es gibt ein Customregister, welches den Wert enthält, mit dem die Portregister gestartet werden. Es handelt sich dabei nur um ein Schreibregister. Zu finden ist dieses ab der Adresse $DFF036.

Gameportstart = $DFF036

Wenn wir dieses Register vor jeder Portabfrage auf Null setzen, haben wir eine 100%ige Portabfrage.

Die Richtungswerte sind für beide Ports gleich. Welcher Wert für welche Richtung steht, können sie der folgenden Tabelle entnehmen.

Richtung:	Wert:	Richtung:	Wert:
oben	$0100	oben/rechts	$0103
unten	$0001	unten/rechts	$0002
rechts	$0003	oben/links	$0200
links	$0300	unten/links	$0301

Es darf natürlich nicht das Beipiellisting fehlen. Abfrage der Richtungen oben und unten:

```
Joy:
  clr.w $dff036                ; Portsignale auf Null
  move.w $dff00c,d0            ; Wert von Port 2 nach D0
oben:
  cmp.w #$0100,d0              ; Richtung oben ?
  bne unten                    ; wenn nicht, dann weiter
  jsr move_oben                ; Figur nach oben bewegen
  bra joy
unten:
  cmp.w #$0001,d0              ; Richtung unten ?
```

```
  bne joy                    ; wenn nicht, wieder von vorne
  jsr move_unten             ; Figur nach unten bewegen
  bra joy
;
;--- Routine fürs bewegen der Figur nach Oben ---
move_oben:
  ;...
  ;...
  rts

;--- Routine fürs bewegen der Figur nach unten ---
move_unten:
  ;...
  ;...
  rts

;--- Ende ---
```

Was uns jetzt noch fehlt, ist die Abfrage der Feuer- und Mausknöpfe. Die folgenden kleinen Beipsiellistings zeigen die Programmierung der Kontrollknöpfe.

Abfrage der Maus/Feuertasten:

Rechte (Port 1):

```
loop:
  btst #10,$dff016
  bne loop
  jsr rechte_taste_port1
  bra loop
```

Linke (Port 1) und Feuerknopf:

```
loop:
  and.b #64,$bfe001
  bne loop
  jsr linke_taste_port1
  bra loop
```

Rechte (Port 2):

```
loop:
  btst #14,$dff016
  bne loop
  jsr rechte_taste_port2
  bra loop
```

Linke (Port 2) und Feuerknopf:

```
loop:
  and.b #128,$bfe001
  bne loop
  jsr linke_taste_port2
  bra loop
```

6.1 Geräuscherzeugung

Da der Amiga bekanntlich ja als Soundmaschine oft bewundert wird, werden wir uns natürlich auch diesem Komplex kurz widmen. Kurz, weil wir hier nicht auf die Programmierung von Musikstücken eingehen werden. Dafür gibt es extra Bücher, die sich speziell mit diesem Thema beschäftigen.

Wir werden nur auf das Abspielen von einzelnen Samples eingehen, also auf die Erzeugung von Soundeffekten. Was ist schon ein Spiel mit noch so schöner Graphik, wenn die Geräuschuntermalung fehlt?

Der Amiga ist, im Gegensatz zu anderen Computern, in der Lage ein Geräusch fast identisch zum Original wiederzugeben. Die meisten Rechner, wie zum Beispiel der Commodore 64, haben vorgegebene Wellenformen, die wiederum in bestimmten Grenzen variierbar sind. Der Amiga kann komplette Wellenformen aufnehmen, über sogenannte Digitalisierer, und fast in der selben Qualität wieder abspielen. Dabei werden die Sampledaten hintereinander als Wörter, die jeweils aus zwei Bytes bestehen, abgespeichert. Um nun ein solches Sample abspielen zu können, muß dem Rechner nur die Anfangsadresse dieser Datenliste übergeben werden und die Anzahl Bytes, die der Sample lang ist.

Die Sampledaten werden über DMA-Kanäle ausgegeben. Damit läuft der Sample unabhängig vom Prozessor, was den Vorteil hat, daß Rechenzeit für den Prozessor eingespart wird. Der Amiga besitzt vier solcher DMA-Soundkanäle, was das Abspielen von vier Samples gleichzeitig ermöglicht.

DMA-Kanal	Registeradresse	Funktion
0	$DFF0A0	Anfangsadresse der Sampledaten 0
1	$DFF0B0	Anfangsadresse der Sampledaten 1
2	$DFF0C0	Anfangsadresse der Sampledaten 2
3	$DFF0D0	Anfangsadresse der Sampledaten 3
0	$DFF0A4	Anzahl Bytes der Sampledaten 0
1	$DFF0B4	Anzahl Bytes der Sampledaten 1
2	$DFF0C4	Anzahl Bytes der Sampledaten 2
3	$DFF0D4	Anzahl Bytes der Sampledaten 3

Soll ein Sample abgespielt werden, muß beachtet werden, daß, wenn das Ende der Sampledaten erreicht wird, automatisch wieder von vorne begonnen wird, die Sampledaten zu lesen und abzuspielen. Damit der Sample wirklich nur einmal abgespielt wird, muß kurze Zeit nach dem Starten die Sampleadresse und -länge auf Null gesetzt werden. Dabei geht der momentane Registerwert nicht verloren, da er intern bearbeitet wird.

Damit wir den Sound auch hören, kann man die Lautstärke für jeden Kanal getrennt einstellen. Dafür sind vier Register vorgesehen, die einen Lautstärkebereich zwischen 0 und 64 zulassen.

DMA-Kanal	Registeradresse	Funktion
0	$DFF0A8	Lautstärke von Kanal 0
1	$DFF0B8	Lautstärke von Kanal 1
2	$DFF0C8	Lautstärke von Kanal 2
3	$DFF0D8	Lautstärke von Kanal 3

Wie schon erwähnt wurde, werden die Sampledaten wordweise gelesen. Dieser Vorgang geschieht in einer bestimmten Geschwindigkeit. Würde man diese Geschwindigkeit ändern, so hätte dies ein veränderten Klang zur Folge, was ja auch logisch ist. Denn ob ich jetzt über die Seiten einer Gitarre schnell oder langsam streiche, der Ton wäre zwar der selbe, aber jeweils in einer anderen Tonhöhe. Das ganze wird als Tonfrequenz (Sampling Rate) bezeichnet.

DMA-Kanal	Registeradresse	Funktion
0	$DFF0A6	Tonhöhe für Kanal 0
1	$DFF0B6	Tonhöhe für Kanal 1
2	$DFF0C6	Tonhöhe für Kanal 2
3	$DFF0D6	Tonhöhe für Kanal 3

Die Tonhöhe läßt sich nach einer Formel berechnen, doch diese ist ziemlich kompliziert. Die meisten Programme zeigen die Tonehöhe auch gleich an, ansonsten kann man ja ein wenig herumexperimentieren.

Hat man soweit alle Vorbereitungen zum Abspielen eines Samples hinter sich, fehlt zum Starten nur noch das Einschalten des entsprechenden Kanals. Dazu muß das jeweilige Bit im DMA-Controllregister gesetzt werden. Für das Einschalten des jeweiligen Kanals, sind die letzen vier Bits im DMA-Controllregister zuständig. Dabei steht das Bit 0 für Kanal 0 usw.

Einschalten von Kanal 0:

```
move.w #$8201,$dff096          ; Kanal 0 ein
```

Ausschalten von Kanal 0:

```
move.w #$0001,$dff096          ; Kanal 0 aus
```

Natürlich haben wir auch für das Abspielen von Samples eine Routine parat. Diese ist jedoch sehr einfach aufgebaut und nicht sehr leistungsfähig. Für das einfache Abspielen reicht sie wohl. Es sollte allerdings darauf geachtet werden, daß Samples nicht zu schnell hintereinander abgespielt werden, da sonst nur ein Knacken zu hören wäre. Am besten ruft man diese Routine vom Interrupt heraus auf.

Routine: PlaySample (A0) (SoundTable)
Parameter: A0 = Zeiger auf SoundTable-Struktur
Erklärung: Spielt einen Sample auf dem gewünschten Kanal einmal ab.
Achtung! Diese Routine wartet nicht, bis der Sample zuende gespielt wurde. Man darf deswegen nicht zu schnell Samples auf dem selben Kanal hintereinander abspielen.

SoundTable-Struktur: (Länge = 16 Bytes)

Offset	Typ	Bezeichnung	Funktion
00	Long	Parameter0Ptr	Zeiger auf Parameterstruktur von Kanal 0
04	Long	Parameter1Ptr	Zeiger auf Parameterstruktur von Kanal 1
08	Long	Parameter2Ptr	Zeiger auf Parameterstruktur von Kanal 2
12	Long	Parameter3Ptr	Zeiger auf Parameterstruktur von Kanal 3
16		END	Ende der SoundTable-Struktur

Die Zahl hinter Parameter gibt den Soundkanal an, über den der Sample abgespielt wird. Soll ein Kanal nicht benutzt werden, muß das Langword (Long) an der entsprechenden Stelle auf Null gesetzt werden.

ParameterXPtr-Struktur: (Länge = 12 Bytes)

Offset	Typ	Bezeichnung	Funktion
00	Long	SampleData	Anfangsadresse der Sampledaten
04	Long	Samplelänge	Anzahl Bytes der Sampledaten
08	Word	Periodendauer	Abtastrate (Tonhöhe) vom Sample
10	Word	Volume	Lautstärke vom Sample (0 bis 64)
12		END	Ende der ParameterXPtr-Struktur

Es folgt das dazugehörige Listing:

```
;--- Spiel einen Soundeffekt nur einmal ab          ---
;--- A0 = Zeiger auf SoundTabelle                   ---
; Aufbau Soundtabelle:          Aufbau ParameterxPointer:
; dc.l parameter0pointer        dc.l Sampledatenpointer
; dc.l parameter1pointer        dc.l Samplelänge in Bytes
; dc.l parameter2pointer        dc.w Periodendauer (Abtastrate)
; dc.l parameter3pointer        dc.w Lautstärke (0 bis 64)
; die Zahl hinter parameter gibt den Soundkanal an, über dem der
; Soundeffekt abgespielt werden soll. Soll ein Kanal nicht benutzt
; werden, muß das Langword an der entsprechenden Stelle auf Null
; gesetzt werden.
;
;
```

```
Playsample:
  clr.w d0                    ; Loopzähler
  clr.w d2                    ; D2 = später DMA-Controll-
                                wert
  lea $dff0a0,a2              ; A2 = Adresse der Soundre-
                                gister Kanal 0
playsample_loop:
  move.l (a0)+,a1             ; Kanäle herausfiltern
  cmp.l #0,a1
  beq playsample_1            ; Kanal benutzen ? (bei Null
                                nicht)
  move.l (a1),a3
  move.l a3,(a2)              ; Audiodatenadresse
  clr.l (a3)                  ; erstes Langword löschen
                                (kein Piepen)
  move.l 4(a1),d3             ; Länge in Bytes
  lsr.l #1,d3                 ; in Words
  move.w d3,4(a2)             ; Audiodatenlänge
  move.w 8(a1),6(a2)          ; Periodendauer
  move.w 10(a1),8(a2)         ; Lautstärke
  move.w #1,d1                ; start Kanal 1
  lsl.w d0,d1                 ; Kanal herausfiltern
  or.w d1,d2                  ; Kanal speichern
playsample_1:
  add.l #$10,a2               ; Soundregister vom näch-
                                sten Kanal
  add.w #1,d0
  cmp.w #4,d0                 ; schon alle 4 Kanäle
  bne playsample_loop
  move.w #$150,d0
  move.w d2,d1                ; D1 = DMA-Wert für aus
  or.w #$8000,d2              ; D2 = DMA-Wert für an
playsample_wait:
  move.w d1,$dff096           ; DMA STOP
  move.w d2,$dff096           ; DMA Start
  dbra d0,playsample_wait
  move.w #$40,d0
playsample_wait1:
```

```
  dbra d0,playsample_wait1
  lea $dff0a0,a0
  move.w #3,d2
playsample_loop2:
  move.w d1,d0
  and.w #1,d0
  tst.w d0
  beq playsample_loop2a
  move.w #1,4(a0)                    ; Audiodatenlänge = 1
  move.l #playsample_space,(a0)      ; Audioadresse
playsample_loop2a:
  add.l #$10,a0
  lsr.w #1,d1
  dbra d2,playsample_loop2
  rts
playsample_space: dc.l  0
;--- Listingende ---
```

Kapitel 7

Grafik-Hardwareprogrammierung

Jetzt endlich kommen wir zu dem Teil, auf den sie bestimmt schon lange warten. Denn was ist ein Spiel ohne Grafik und Animation? Sie als angehender Spieleprogrammierer und stolzer Besitzer eines Amiga werden in diesem Kapitel alles Wissenswerte über die Grafikprogrammierung erfahren. Dabei wird kein Vorwissen oder derartiges vorausgesetzt. Wir werden auf die einfache Erzeugung von Objekten bis hin zu komplexen Animationen eingehen. Natürlich ordentlich mit Beispiellistings versehen.

Um überhaupt gute Animationen erzeugen zu können, wurden sämtliche Routinen über die Hardware programmiert. Sie brauchen jetzt aber nicht zurückschrecken, denn alle Beispielroutinen sind einfach zu handhaben und erledigen für sie die ganze komplizierte Hardwareprogrammierung. Für die Profis wurden die Beispiellistings gut dokumentiert und sind somit leicht nachzuvollziehen.

7.1 Der Blitter

Das wohl wichtigste Bauteil im Amiga für die Spieleprogrammierung ist, ohne zu übertreiben, der Blitter, denn dieser Grafik-Chip ist in der Lage 1 Million Punkte in einer Sekunde zu verschieben. Man kann nicht nur Daten hin und her verschieben, sondern auch auf verschiedene Arten miteinander verknüpfen. Der Blitter ist in der Lage, die eben aufgezählten Funktionen mit drei Quellen gleichzeitig durchzuführen, wobei das Ergebnis ab einer bestimmten Zieladresse abgelegt wird.

Der Blitter wird auch für die Co- und Decodierung der Diskdaten benutzt. Doch wir werden hier nur auf bestimmte Verknüpfungen eingehen, die für die Darstellung von Objekten erforderlich sind. Diese Objekte werden auch Blitterobjekte, kurz - BOBs -, genannt.

Die Programmierung des Blitters ist sehr kompliziert. Doch wie schon erwähnt wurde, werden sie damit ja nicht konfrontiert. Das erledigen dann zur gegebener Zeit die Beispielroutinen. Auch das System stellt uns einige Routinen für die Darstellung von Objekten zur Verfügung, doch sind diese viel zu langsam und umständlich.

Wir werden uns für alle nötigen Funktionen, wie Darstellung, Kollisionsabfrage usw., selbst Routinen programmieren und somit ein ganz neues, eigenes und erweiterungsfähiges Grafiksystem zusammenstellen. Am Ende des Kapitels werden sie eine Sammlung von Routinen besitzen, die sich mit den schnellsten und komfortabelsten die je auf dem Amiga programmiert wurden, messen können. Damit besitzen sie die Basis zur Programmierung von Superspielen, die fast an die Qualität von Automaten herankommen. Natürlich bleibt es letztendlich an ihnen, was sie daraus machen.

7.2 Grafiken kopieren

Die einfachste Funktion des Blitters ist der Datentransfer. Das bedeutet, daß Daten, die ab einer bestimmten Adresse im Speicher liegen, wir nennen sie mal Quelldaten, zu einer anderen Adresse, die sogenannte Zieladresse, hin verschoben werden. Dabei können wir maximal einen Speicherblock der Größe 1024 X 1024 Pixel verschieben. Das wären 128 KByte oder anders ausgedrückt 131072 Bytes. Vergleichbar mit 2,5 Bildschirmen der Größe 256 X 320 mit einer Tiefe von 5 Bitplanes.

Nehmen wir mal an, wir wollen nur einen bestimmten Teil aus einer Grafik herauskopieren. Die Teilgrafik soll 32 Pixel breit sein, also 2 Words bzw. 4 Bytes und die eigentliche Grafik besitzt eine Größe von 64 Pixel (= 4 Words , = 8 Bytes). Der Blitter müßte also

am Ende jeder Grafikzeile 32 Pixel überspringen, bevor er die nächste Zeile kopiert. Für diesen Zweck sind bestimmte Blitterregister vorgesehen, die die Differenz zwischen Grafik und Teilgrafik beinhalten. Diese Differenz wird als Modulo bezeichnet und berechnet sich wie folgt:

(Grafik - Teilgrafik) / 16 * 2 = Modulo in Bytes

Beispiel: (32 - 16) / 16 * 2 = 2 Bytes

Der Modulowert muß immer eine gerade Anzahl von Bytes besitzen. Deswegen können auch keine Grafiken der Größe von 14 Pixeln verschoben werden. Unmöglich ist das natürlich auch nicht. Denn das erste und letzte Word (= 16 Pixel) einer Zeile wird mit einer Maske belegt. Normalerweise steht dort der Wert $FFFF. Es werden also alle Pixel übernommen. Da es sich um 1 Word handelt, also um 16 Bits, steht jedes Bit für ein Pixel. Will man jetzt nur die ersten 14 Bits übernehmen, so setzt man auch nur die ersten 14 Bits, wodurch die erste Maske einen Wert von %0011111111111111 enthält. Das zweite Word enthält aber weiterhin den Wert $FFFF. Es sei denn, die Grafik ist nur 16 Pixel groß, dann fallen beiden Masken zusammen und die zweite Maske müßte den selben Wert enthalten. Diese Register werden First-Mask und LastMask genannt.

Man besitzt auch die Möglichkeit, mit dem Blitter den Quelldatenbereich vor der Ausgabe bis zu 16 Bits nach rechts zu verschieben. Dadurch ist man in der Lage eine Grafik an eine beliebige Position auf den Bildschirm zu verschieben. Zu beachten ist dabei, daß die Punkte, die herausgeschoben, nicht etwa gelöscht werden, sondern an den Anfang der gleichen Zeile erscheinen. Um diesen unschönen Effekt zu vermeiden, definiert man die Grafik, welche verschoben werden soll, ein Word breiter. Dieses letzte Word muß allerdings Null sein. Dann werden die Bits des letzten Words an den Anfang der Zeile verschoben. Da aber keine vorhanden sind, erscheinen auch keine.

Es wurde ja schon erwähnt, daß mit dem Blitter logische Verknüpfungen durchgeführt werden können. Von den zahlreichen Verknüpfungen wollen wir allerdings nur auf zwei eingehen. Die eine läßt eine Verschiebung zu, bei der alle Bits gesetzt und gelöscht sind. Die andere eine, wo nur die gesetzten Bits verschoben werden. Dies würde einem logischen ODER entsprechen.

Die Routine, welche die nötigen Schritte für uns an die Blitterhardware übernimmt, nennt sich CopyGrafik ().

Routine: CopyGrafik (A0) (BlitterArgs)
Parameter: A0 = Zeiger auf Blitterargumentenliste, welche alle nötigen Informationen für den Blitter enthält.
Erklärung: Kopiert Daten von einem Quellbereich zu einem Zielbereich.

BlitterArgs-Struktur: (Länge = 22 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
00	Long	Quelle	Anfangsadresse des Quelldatenbereichs, welcher verschoben werden soll.
04	Long	Ziel	Anfangsadresse des Zielbereichs, zu dem die Daten, auf die Quelle zeigt, hinverschoben werden sollen.
08	Word	QModulo	Modulowert der Quelle in Bytes
10	Word	ZModulo	Modulowert des Ziels in Bytes
12	Word	WBreite	Breite der Grafik, welche verschoben werden soll, in Words.
14	Word	Höhe	Höhe der Grafik in Zeilen
16	Byte	Art	Art des Datentransfers: 1 = nur gesetzte Bits werden kopiert 0 = alle Bits werden kopiert

17	Byte	QShift	Anzahl Punkte, die der Quelldatenbereich vor der Ausgabe nach rechts geschoben werden soll. Werte zwischen 0 und 15 sind erlaubt.
18	Word	FirstMask	erste Maske der Quelle (16 Bits)
20	Word	LastMask	letzte Maske der Quelle (16 Bits)
22		END	Ende der BlitterArgs-Struktur

Hier das dazugehörige Listing der Routine CopyGrafik ():

```
;--- Routine zum Copieren von Grafik mit dem Blitter ----
;--- A0 = Blitterargs ---
CopyGrafik:
  move.l #$dff000,a6
  move.l (a0),$50(a6)          ; Anfangsadresse von Quelle
                                 A
  move.l 4(a0),$54(a6)         ; Anfangsadresse von Ziel  D
  move.w #$0900,d5             ; USE nur Quelle A und Ziel D
  move.b 17(a0),d6             ; Shiftwert für Quelle A holen
  lsl.w #8,d6                  ; korrekten Shiftwert ermit-
                                 teln
  lsl.w #4,d6
  add.w d6,d5                  ; ShiftWert (ASHx) zu
                                 BLTCON0 addieren
  tst.b 16(a0)                 ; Was für eine Art liegt vor?
  bne cg1
  add.w #$00f0,d5              ; Art = 0 (alle Bits kopieren)
                               ; Minterm: A = D
  bra cg2
cg1:                           ; Art = 1 (nur gesetzte Bits
                                 kopieren)
  move.l 4(a0),$4c(a6)         ; Anfangsadresse von Quelle
                                 B = Ziel D
```

```
  add.w #$04fc,d5          ; USE auch Quelle B (für
                           ; ODERverknüpfung mit A)
                           ; Minterm: A + B = D
cg2:
  move.w d5,$40(a6)        ; BLTCON0: mit entspre-
                           ; chendem Wert laden
  clr.w $42(a6)            ; BLTCON1: no Shift-Quelle
                           ; B + aufsteigende Adr.
  move.l 18(a0),$44(a6)    ; First/Last Mask (alle Bits
                           ; übernehmen)
  move.w 8(a0),$64(a6)     ; Modulowert von Quelle A
  move.w 10(a0),$62(a6)    ; Modulowert von Quelle B
  move.w 10(a0),$66(a6)    ; Modulowert von Ziel D
  move.w 12(a0),d5         ; Breite in Words
  move.w 14(a0),d6         ; Höhe in Pixel
  and.w #$3ff,d6
  and.w #$3f,d5
  lsl.w #6,d6              ; D5 = breite in Words
  add.w d6,d5              ; D5 ist jetzt Blittersize
  move.w d5,$58(a6)        ; BLTSIZE und Blitteroperati-
                           ; on starten
copyg_wait:
  btst #14,$2(a6)          ; auf Blitterende warten
  bne copyg_wait
  rts
;
;--- Listingende ----
```

Das folgende Beispielprogramm erzeugt einen Screen über die Hardware und kopiert eine Grafik in die BitMap. Durch Drücken der rechten Maustaste kann das Programm beendet werden.

Da wir nicht die System-Interrupts ausschalten, müssen wir dem System verbieten auf den Blitter zugreifen zu können. Ansonsten kann es zu Abstürzen führen, wenn wir gleichzeitig mit dem System auf den Blitter zugreifen. Mit der Routine OwnBlitter () verbieten wir dem System sämtliche Blitteroperationen. Das Gegen-

stück zu OwnBlitter () ist DisOwnBlitter (), welche die Blitteroperationen wieder erlaubt. Beide Routinen benötigen keine Parameter und sind in der Graphics.library zu finden.

Das folgende Listing ist so gut dokumentiert, daß keine weiteren Informationen erforderlich sind.

```
;--- Erzeugt einen Screen über die Hardware      ---
;--- Kopiert eine Grafik mit CopyGrafik ()       ---
;--- und auf rechte Maustaste erscheint alte     ---
;--- Copperliste und Programm wird beendet       ---
;--- Programmname = Copy-Screen                  ---
;
  move.l 4,a6                 ; ExecBasis
  lea gfxname,a1              ; Libraryname
  clr.l d0                    ; Version = 0
  jsr -552(a6)                ; OpenLibrary ()
  move.l d0,gfxbase           ; graphics.basis
;
  lea rastport,a1             ; RastPort-Struktur
  move.l gfxbase,a6           ; graphics.basis
  jsr -198(a6)                ; InitRastPort ()
;
  lea bitmap,a0               ; BitMap-Struktur
  move.l #3,d0                ; Depth = 3
  move.l #320,d1              ; 320 breit
  move.l #256,d2              ; 256 hoch
  move.l #1,d3                ; Speicher reservieren
  bsr initbitmap              ; BitMap init.
;
  lea bitmap,a0               ; BitMap-Struktur
  lea rastport,a1             ; RastPort-Struktur
  move.l a0,4(a1)             ; BitMap in RastPort einhän-
                                gen
;
  lea colortable,a0           ; Farbtabelle
  lea colorpuffer,a1          ; Copperpuffer
  bsr initcolor               ; Farbregister init.
```

```
;
  lea bitmap,a0                    ; BitMap-Struktur
  lea copperpuffer,a1              ; Copperpuffer
  clr.l d0                         ; Modus = Normal
  bsr initcoppermap                ; Customregister init.
;
  move.l #copperlist,$dff080       ; Copperlistadresse
  clr.w $dff088                    ; Copperliste starten
;
  move.w #$20,$dff096              ; Sprite-DMA aus
  move.l #spritedatas,$dff120      ; Sprite 0 (Mauszeiger) aus
;
;--- Blitterfunktionen dem System verbieten ---
  move.l gfxbase,a6
  jsr -456(a6)                     ; OwnBlitter ()
;
;--- Grafik kopieren ---
  lea blitterargs,a0               ; A0 = Blitterargsadresse
  lea bitmap,a1                    ; A1 = BitMap-Struktur
  move.l 8(a1),a2                  ; A2 = Plane1-Adresse
  move.w (a1),d0                   ; Screenbreite in Bytes
  sub.w #6,d0                      ; minus Quellbreite (3 Words)
  move.w d0,10(a0)                 ; ZModulo
  clr.w 8(a0)                      ; QModulo
  move.l #quelle,(a0)              ; Quelle
  move.l a2,4(a0)                  ; Ziel
  move.w #3,12(a0)                 ; WBreite
  move.w #4,14(a0)                 ; Höhe
  clr.b 16(a0)                     ; Art = alle Bits kopieren
  clr.b 17(a0)                     ; QShift
  move.w #$ffff,18(a0)             ; FirstMask
  move.w #$ffff,20(a0)             ; LastMask
  bsr copygrafik                   ; Grafik kopieren
;
;--- Blitter wieder freigeben ---
  move.l gfxbase,a6
  jsr -462(a6)                     ; DisownBlitter ()
;
```

```
  lea demotext,a0                ; Textadresse
  lea rastport,a1                ; RastPort-Struktur
  move.l gfxbase,a6              ; graphics.basis
  clr.l d0                       ; X-Position
  move.l #20,d1                  ; Y-Position
  bsr printtext                  ; Text printen
;
;--- Maustaste abfragen ---
main:
  btst #10,$dff016               ; Rechte Maustaste ?
  bne main
;
;--- Programm beenden ---
  move.l gfxbase,a0              ; graphics.basis
  move.l 38(a0),$dff080          ; alte Copperlistadresse
  clr.w $dff088                  ; und starten
;
  move.w #$8220,$dff096          ; Sprite-DMA an
;
  lea bitmap,a0                  ; BitMap-Struktur
  bsr clearbitmap                ; BitMap freigeben
;
  move.l gfxbase,a1              ; graphics.basis
  move.l 4,a6                    ; exec.basis
  jsr -414(a6)                   ; CloseLibrary ()
;
  rts                            ; Programmende


;--- Gibt einen Text auf dem Bildschirm aus,      ---
;--- der mit einem Nullbyte endet                 ---
;--- Parameter:   A0 = Text, A1 = RastPort,       ---
;---              A6 = Graphics-Basis             ---
;---              D0 = X-Pos., D1 = Y-Position    ---
PrintText:
  clr.w d2                       ; Zähler für Anzahl Zeichen
  move.l a0,a2
```

```
pt_loop:
  tst.b (a2)+
  beq pt_loop_end
  add.w #1,d2
  cmp.w #100,d2                ; Max. 80 Zeichen
  bne pt_loop
pt_loop_end:
  tst.w d2                     ; Kein Text ?
  beq pt_end
  movem.l a0-a1/a6/d2,-(sp)    ; Parameter retten
  jsr -240(a6)                 ; MOVE () - Position setzen
  movem.l (sp)+,a0-a1/a6/d0    ; Parameter holen
  jsr -60(a6)                  ; TEXT () - Text printen
pt_end:
  rts

;--- BitMap-Pointer in CopperList übertragen    ---
;--- A0 = BitMap ; A1 = CopperPuffer            ---
;--- D0 = Modus                                 ---
InitCopperMap:
  move.w (a0),d1               ; Bytes pro Zeile nach D1
  sub.w #40,d1                 ; minus 40 Bytes (320 Pixel)
  cmp.w #$8000,d0              ; Modus = Hires ?
  bne icm_1
  sub.w #40,d1                 ; nochmal minus 40 Bytes
                               (insgesamt -640 Pixel)
icm_1:
  cmp.w #$04,d0                ; Modus = Interlace ?
  bne icm_2
  sub.w #40,d1                 ; auch minus 40 Bytes (ins-
                               gesamt -640 Pixel)
icm_2:
  move.w #$0108,(a1)+          ; BPL1MOD
  move.w d1,(a1)+              ; Modulo-Wert ungerade Pla-
                               nes
  move.w #$010a,(a1)+          ; BPL2MOD
  move.w d1,(a1)+              ; Modulo-Wert gerade Planes
  move.b 5(a0),d1              ; Depth nach D1
```

```
  lsl.w #8,d1                ; Korrekten
  lsl.w #5,d1                ; Depth-Wert
  lsr.w #1,d1                ; ermitteln
  add.w #$0200,d1            ; Color setzen
  add.w d0,d1                ; Modus setzen
  move.w #$0100,(a1)+        ; BPLCON0
  move.w d1,(a1)+            ; setzen
  moveq #5,d0                ; max. Tiefe minus 1
  move.w #$00e0,d1           ; Hi-Word vom ersten Map-
                               Pointer
  add.l #8,a0                ; Erster Pointer-Zeiger
icm_loop:
  move.w d1,(a1)+            ; HI-Word vom Pointer
  move.w (a0)+,(a1)+         ; setzen
  add.w #2,d1                ; Lo-Word ermitteln
  move.w d1,(a1)+            ; und
  move.w (a0)+,(a1)+         ; setzen
  add.w #2,d1                ; Hi-Word ermitteln
  dbra d0,icm_loop
  rts

;--- ColorMap in CopperListe übertragen      ---
;--- A0 = ColorTable ; A1 = CopperPuffer     ---
InitColor:
  move.w #$180,d1            ; erstes Farbregister
  move.w #31,d0              ; Anzahl Farben
ic_loop:
  move.w d1,(a1)+            ; Farbregister in CopperList
  move.w (a0)+,(a1)+         ; dann der Farbwert in Cop-
                               perList
  add.w #2,d1                ; nächstes Farbregister
  dbra d0,ic_loop
  rts

;--- BitMap-Struktur initialisieren und Speicher  ---
;--- für Maps reserverieren                       ---
;--- D0 = Depth, D1 = Width, D2 = Height,         ---
;--- D3 = Memory, A0 = BitMap                     ---
```

```
InitBitMap:
  move.w d1,d4              ; Breite nach D4
  and.w #15,d4              ; Rest ermitteln
  tst.w d4                  ; Rest vorhanden ?
  beq ibm_1                 ; Wenn nicht, dann weiter -
                              sonst Error
  move.l #-1,d0             ; Error: Breite nicht durch 16
                              teilbar
  rts
ibm_1:
  lsr.w #4,d1               ; Breite durch 16 teilen
  lsl.w #1,d1               ; mal 2 = Anzahl Bytes einer
                              Zeile
  move.w d1,(a0)            ; und in BitMap-Struktur
                              speichern
  move.w d2,2(a0)           ; Höhe in BitMap-Struktur
                              speichern
  move.w d0,4(a0)           ; Anzahl Planes speichern
  tst.w d3                  ; Muß Speicher für BitMaps
                              reserviert werden ?
  bne ibm_2                 ; Wenn nicht, dann Ende,
                              sonst weiter
  clr.l d0                  ; No Errors
  rts                       ; Ende
ibm_2:
  move.w d0,d4              ; Depth nach D4
  sub.w #1,d4               ; minus 1
  lea 8(a0),a1              ; BitMap nach a1
ibm_clear_loop:             ; Pointer-Zeiger
  clr.l (a1)+               ; löschen
  dbra d4,ibm_clear_loop
  mulu d2,d1                ; BytePerRow x Höhe = Size
                              von einer Map
  move.w d0,d2              ; D2 = Anzahl Planes
  move.l d1,d0              ; D0 = ByteSize
  move.l #$10002,d1         ; D1 = CHIP + FreeMem
  sub.w #1,d2               ; Anzahl Planes minus 1, we-
                              gen DBRA
```

```
  add.l #8,a0                    ; Anfang BitMap-Zeiger
  move.l a0,a2                   ; auch nach A2
ibm_loop:
  move.l #$4,a6
  move.l (a6),a6
  movem.l d0-d2/a0-a2,-(sp)
  jsr -198(a6)                   ; AllocMem
  tst.l d0                       ; konnte Speicher reserviert
                                   werden ?
  bne ibm_l1                     ; Wenn ja, dann weiter
  movem.l (sp)+,d0-d2/a0-a2
ibm_free_loop:
  move.l (a2),a1                 ; Zeiger auf BitMap holen
  cmp.l #0,a1                    ; Null ?
  bne ibm_l2                     ; wenn ja, dann Ende
  move.l #-2,d0                  ; Error
  rts                            ; Ende
ibm_l2:
  clr.l (a2)+
  movem.l d0/a2,-(sp)
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                   ; FreeMem
  movem.l (sp)+,d0/a2
  bra ibm_free_loop
ibm_l1:
  move.l d0,d5                   ; Zeiger auf BitMap nach D5
  movem.l (sp)+,d0-d2/a0-a2
  move.l d5,(a0)+
  dbra d2,ibm_loop
  clr.l d0
  rts

;--- Speicher der Maps wieder freigeben          ---
;--- in einer BitMap-Struktur                    ---
;--- A0 = BitMap                                 ---
ClearBitMap:
  clr.w d1
```

```
  move.b 5(a0),d1              ; Depth nach D1
  sub.w #1,d1
  move.w (a0),d0               ; BytePerRow
  mulu 2(a0),d0                ; mal Höhe = MapSize
  add.l #8,a0
  move.l #$4,a6
  move.l (a6),a6
cbm_loop:
  move.l (a0),a1               ; Map-Pointer
  cmp.l #0,a1
  beq cbm_l1
  clr.l (a0)+
  movem.l d0-d1/a0/a6,-(sp)
  jsr -210(a6)                 ; FreeMem
  movem.l (sp)+,d0-d1/a0/a6
cbm_l1:
  dbra d1,cbm_loop
  rts

;--- Routine zum Copieren von Grafik mit dem Blitter          ---
;--- A0 = Blitterargs                                         ---
CopyGrafik:
  move.l #$dff000,a6
  move.l (a0),$50(a6)          ; Anfangsadresse von Quelle
                               A
  move.l 4(a0),$54(a6)         ; Anfangsadresse von Ziel D
  move.w #$0900,d5             ; USE nur Quelle A und Ziel D
  move.b 17(a0),d6             ; Shiftwert für Quelle A holen
  lsl.w #8,d6                  ; korrekten Shiftwert ermit-
                               teln
  lsl.w #4,d6
  add.w d6,d5                  ; ShiftWert (ASHx) zu
                               BLTCON0 addieren
  tst.b 16(a0)                 ; Was für eine Art liegt vor?
  bne cg1
  add.w #$00f0,d5              ; Art = 0 (alle Bits kopieren)
                               ; Minterm: A = D
  bra cg2
```

```
cg1:                                ; Art = 1 (nur gesetzte Bits
                                      kopieren)
  move.l 4(a0),$4c(a6)              ; Anfangsadresse von Quelle
                                      B = Ziel D
  add.w #$04fc,d5                   ; USE auch Quelle B (für
                                      ODERverknüpfung mit A)
                                    ; Minterm: A + B = D
cg2:
  move.w d5,$40(a6)                 ; BLTCON0: mit entspre-
                                      chendem Wert laden
  clr.w $42(a6)                     ; BLTCON1: no Shift-Quelle
                                      B + aufsteigende Adr.
  move.l 18(a0),$44(a6)             ; First/Last Mask (alle Bits
                                      übernehmen)
  move.w 8(a0),$64(a6)              ; Modulowert von Quelle A
  move.w 10(a0),$62(a6)             ; Modulowert von Quelle B
  move.w 10(a0),$66(a6)             ; Modulowert von Ziel D
  move.w 12(a0),d5                  ; Breite in Words
  move.w 14(a0),d6                  ; Höhe in Pixel
  and.w #$3ff,d6
  and.w #$3f,d5
  lsl.w #6,d6                       ; D5 = Breite in Words
  add.w d6,d5                       ; D5 ist jetzt Blittersize
  move.w d5,$58(a6)                 ; BLTSIZE und Blitteroperati-
                                      on starten
copyg_wait:
  btst #14,$2(a6)
  bne copyg_wait
  rts

;--- Parameter ---
gfxbase: dc.l 0                     ; graphics.basis
gfxname: dc.b "graphics.library",0  ; Libraryname
  even
bitmap: blk.b 40,0                  ; BitMap-Struktur
rastport: blk.b 100,0               ; RastPort-Struktur
colortable:
  dc.w 0,1000,2000,3000
```

```
  blk.w 28,235                    ; 32 Farben
spritedatas: blk.l 3,0
copperlist:                       ; ab hier Copperliste
  dc.w $8e,$3081                  ; DIWSTRT
  dc.w $90,$30c1                  ; DIWSTOP
  dc.w $92,$38                    ; DDFSTRT
  dc.w $94,$d0                    ; DDFSTOP
colorpuffer:
  blk.b 128,0                     ; Farbregister
copperpuffer:
  blk.b 60,0                      ; Screen-Customregister
  dc.l $fffffffe                  ; Copperlistende
demotext:
  dc.b "Grafik kopieren mit dem Blitter",0
  even
blitterargs:
  blk.b 22,0                      ; 22 Bytes BlitterArgs
Quelle:                           ; Höhe = 4 , Breite = 3 Words
  dc.w %0011111000000001,%1000000001111100,$0
                                  ; letztes Word = 0
  dc.w %0011111000000011,%1100000001111100,$0
                                  ; letztes Word = 0
  dc.w %0011000000000111,%1110000000001100,$0
                                  ; letztes Word = 0
  dc.w %0011111111111111,%1111111111111100,$0
                                  ; letztes Word = 0
  END
;--- Listingende ---
```

7.3 Bitmaps kopieren

Für das flackerfreie Darstellen von Objekten ist es notwendig, den Hintergrund, worauf das Objekt gezeichnet werden soll, in einen Puffer zu retten. Dabei stehen einem mehrere Möglichkeiten offen. Wir wollen hier aber nur auf eine eingehen. Die anderen werden im Kapitel 7.10 beschrieben.

Das ganze Prinzip der flackerfreien Bewegung von Objekten beruht darauf, daß diese im Hintergrund aufgebaut werden. Das heißt, daß der eigentliche Hintergrund nicht nur einmal vorhanden ist, sondern mindestens zweimal oder wie in unserem Beispiel sogar dreimal. Dabei werden die Objekte in die zweite Bitmap (Hintergrund) gezeichnet, welche jedoch nicht sichtbar ist. Jetzt wird diese aktiviert und die Objekte werden blitzschnell dargestellt. Was jetzt nur noch fehlt, ist das Restaurieren des Hintergrundes. Denn die jetzt unsichtbare Bitmap enthält ja die alte Position der Objekte. Würden wir jetzt wieder die Objekte nach beschriebener Methode erscheinen lassen, hätte dies ein verschwommenes Objekt zur Folge. Deswegen legen wird den Hintergrund einfach dreimal im Speicher ab, und kopieren die dritte Bitmap nach dem aktivieren der unsichtbaren Bitmap, in die jetzt unsichtbare. Damit befindet sich der Hintergrund immer in seiner Ausgangslage. Es gibt zwar auch andere Möglichkeiten, aber wie schon erwähnt wurde, wird darauf später eingegangen. Diese Methode ist auf den ersten Blick sehr speicherraubend, doch das täuscht. Wenn viele Objekte verwaltet werden sollen, erweist sie sich sogar als speichersparend. Außerdem ist sie sehr einfach.

Im vorangegangen Abschnitt waren wir auf das einfache Kopieren von Grafiken mit dem Blitter eingegangen. Wir könnten jetzt natürlich jede einzelne Bitmap in der BitMap-Struktur mit der Routine "CopyGrafik ()" kopieren. Doch das wäre viel zu umständlich. Nebenbei ist es auch nicht gerade sehr schnell. Viel einfacher und schneller würde es mit einer Routine gehen, welche speziell nur BitMaps kopiert.

Sie werden es sich sicherlich schon denken können. Auch für diesen Zweck habe ich eine Routine geschrieben. Die Funktion, die sich sinnvollerweise CopyBitMap () nennt, benötigt als Parameter nur die Anfangsadresse der Quell- und Ziel-Bitmap-Struktur. Den Rest erledigt die Routine. Das Listing ist sehr kurz und da die Bit-Map natürlich mit dem Blitter verschoben wird, auch sehr schnell.

Routine: CopyBitMap (A0,A1) (Quell-BitMap,Ziel-BitMap)
Parameter: A0 = Adresse der Quell-BitMap-Struktur, welche verschoben werden soll.
A1 = Adresse der Ziel-BitMap-Struktur, wo die Quell-BitMap-Struktur hinverschoben werden soll.
Erklärung: Diese Routine kopiert die BitMap-Struktur auf die A0 (Quelle) zeigt, in die BitMap-Struktur auf die A1 (Ziel) zeigt. Die BitMap darf maximal 1024 Pixel breit und 1024 Zeilen hoch sein.

Hier das dazugehörige Listing:

```
;--- Routine zum Copieren von Grafik mit dem Blitter          ---
;--- A0 = Quell-BitMap ; A1 = Ziel-BitMap                     ---
CopyBitMap:
  move.l #$dff000,a6              ; Custom-Basisadresse
  move.l #$09f00000,$40(a6)       ; USE A und D
                                  ; Minterm: A = D
  move.l #-1,$44(a6)              ; First/Last Mask (alle Bits
                                    übernehmen)
  clr.l $64(a6)                   ; Modulowert von Quelle A
  move.w (a0),d5                  ; Breite in Bytes
  lsr.w #1,d5                     ; jetzt in Words
  move.w 2(a0),d6                 ; Höhe in Pixel
  and.w #$3ff,d6                  ; Blittersize errechnen
  lsl.w #6,d6                     ; D5 = Breite in Words
  and.w #$3f,d5                   ; D6 = Höhe in Pixel
  add.w d6,d5                     ; D5 ist jetzt Blittersize
  clr.w d0
  move.b 5(a0),d0                 ; Tiefe
```

```
  subq #1,d0                ; minus 1
  add.l #8,a0
  add.l #8,a1
cbmap_loop:
  move.l (a0)+,$50(a6)      ; Anfangsadresse von Quelle
                              A
  move.l (a1)+,$54(a6)      ; Anfangsadresse von Ziel D
  move.w d5,$58(a6)         ; BLTSIZE und Blitteroperati-
                              on starten
cbmap_wait:
  btst #14,$2(a6)           ; Warten bis Blit fertig
  bne cbmap_wait
  dbra d0,cbmap_loop
  rts
;--- Listingende ----
```

Diese Routine kopiert blitzschnell Grafiken. Doch was ist, wenn wir die Grafik nicht mehr haben wollen, sondern gerne auf einem leeren Hintergrund Text ausgeben möchten?

Was wir dazu brauchen, ist eine Funktion, die den Hintergrund löscht. Auf dem Commodore 64 reichte das Printen eines inversen Herzens und der Bildschirm war frei. Der Amiga bietet uns dafür leider nicht so eine einfache Methode. Deswegen folgt eine Routine, welche mit Hilfe des Blitters den Bildschirm löscht. Doch diese Funktion kann einiges mehr. Man kann ihr ein 16-Bit großes Muster in dem Datenregister D0 übergeben. Mit diesem Muster wird dann der ganze Bildschirm gefüllt. Setzt man das Datenregister auf den Wert Null, so wird der Bildschirm gelöscht. Das ganze läuft so superschnell ab, daß es für das menschliche Auge gar nicht wahrnehmbar ist und damit sofort erscheint.

Routine: FillBitMap (D0,A1) (Muster,BitMap)
Parameter: D0 = Füllmuster (16 Bit bzw. 1 Word groß) mit dem die BitMap gefüllt werden soll.
A1 = Adresse der BitMap-Struktur, welche mit dem Muster gefüllt werden soll.
Erklärung: Diese Routine füllt eine BitMap-Struktur auf die A1 zeigt, mit einem 16-Bit großen Muster, welches im Datenregister D0 übergeben wird. Die Ausmaße sind die selben wie bei CopyBitMap ().

Listing:

```
;--- Routine zum Füllen der Bitmap mit dem Blitter      ---
;--- D0 = FillMuster, A1 = BitMap                       ---
FillBitMap:
  move.l #$dff000,a6
  move.w d0,$74(a6)              ; Fillmuster
  move.l #$01f00000,$40(a6)      ; USE A und D
                                 ; Minterm: A = D
  move.l #-1,$44(a6)             ; First/Last Mask
                                   (alle Bits übernehmen)
  clr.l $64(a6)                  ; Modulowert von Quelle A
  move.w (a1),d5                 ; Breite in Bytes
  lsr.w #1,d5                    ; jetzt in Words
  move.w 2(a1),d6                ; Höhe in Pixel
  and.w #$3ff,d6                 ; Blittersize errechnen
  lsl.w #6,d6                    ; D5 = Breite in Words
  and.w #$3f,d5                  ; D6 = Höhe in Pixel
  add.w d6,d5                    ; D5 ist jetzt Blittersize
  clr.w d0
  move.b 5(a1),d0                ; Tiefe
  subq #1,d0                     ; minus 1
  add.l #8,a1
fbmap_loop:
  move.l (a1)+,$54(a6)           ; Anfangsadresse von Ziel D
  move.w d5,$58(a6)              ; BLTSIZE und Blitteroperati-
                                   on starten
```

```
fbmap_wait:
  btst #14,$2(a6)                ; Warten bis Blit fertig
  bne fbmap_wait
  dbra d0,fbmap_loop
  rts
;--- Listingende ----
```

7.4 BOBs - Blitterobjekte

Der Amiga ist in Lage, über seine Hardware bis zu acht Sprites darzustellen. Ein Sprite ist ein Gebilde aus mehreren Grafikpixeln. Der Mauszeiger ist zum Beispiel auch ein Sprite. Diese Sprites werden mit Hilfe der Hardware über eigene DMA-Kanäle auf dem Bildschirm ausgegeben. Diese Sprites sind jedoch in vieler Hinsicht eingeschränkt. Sie sind nicht besonders breit und können auch nicht viele Farben besitzen. Damit sind sie für die Spieleprogrammierung nicht gerade besonders geeignet.

Für ein gutes Spiel sind schon mehr als acht Objekte erforderlich. Natürlich sollten diese in ihren Ausmaßen nicht begrenzt sein. Diese Aufgaben erfüllen die sogenannten BOBs bzw. Blitter-objekte. Ein BOB ist ein Blitter-Objekt, welches, wie der Name schon sagt, mit dem Blitter erzeugt wird. Dabei handelt es sich um eine Grafik, die mit dem Blitter blitzschnell in den Hintergrund kopiert werden kann. Ein Blitter-Objekt kann die maximale Anzahl von Farben wie der Hintergrund besitzen. Auch wird dieser in der selben Auflösung dargestellt. Allerdings ist das Blitter-Objekt keinen Ausmaßen unterlegen.

Würde man jetzt aber einfach eine solche Teilgrafik (BOB) mit dem Blitter in den Hintergrund kopieren, so würde dieser ja verlorengehen. Es ist deswegen notwendig, die Hintergrundgrafik mit den gleichen Ausmaßen wie das Blitter-Objekt in einen Puffer zu retten. Will man jetzt einen BOB über den Hintergrund bewegen, so muß als erstes der Hintergrund, der vorher gerettet wurde, in den Hintergrund an seiner alten Position zurückgeschrieben werden.

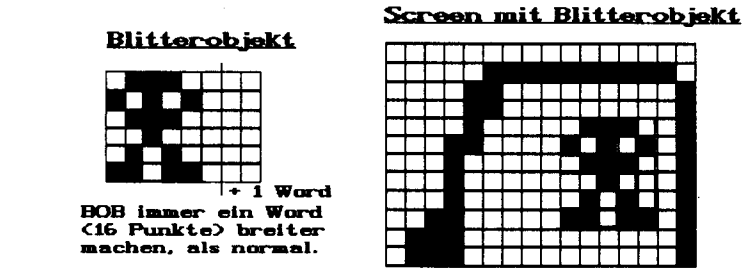
Als zweites muß jetzt der Hintergrund an den neuen Positionen des BOBs gerettet werden, damit dieser später auch wieder hergestellt werden kann. Jetzt erst können sie das eigentliche Blitter-Objekt in den Hintergrund kopieren. All diese Schritte müssen jedesmal durchgeführt werden, wenn sie das Blitter-Objekt bewegen möchten.

Bei diesem Aufwand spielt der Faktor Zeit eine große Rolle. Die Geschwindigkeit, welche für das Kopieren der Grafiken erforderlich ist, kann natürlich nicht mit dem Prozessor erreicht werden. Dafür wird, wie kann es auch anders sein, der Blitter eingesetzt.

Kurzbeschreibung von BOBs:

Ausmaße : beliebig (maximal 1024 X 1024 Pixel)
Farben : maximal so viele wie der Hintergrund
Anzahl : beliebig viele (bis Speicher voll ist)
Auflösung : dieselbe wie der Hintergrund

Aufbau von einem BOB



Mit dem Malprogramm Deluxe-Paint (oder auch andere) kann man Teile der Hintergrundgrafik ausschneiden und diese als Brushes auf Diskette abspeichern. Diese Brushes sind nichts anderes als Blitter-Objekte. Nur werden diese in einem bestimmten Format abgespeichert. Wie wir dieses Format in für uns brauchbare Daten umwandeln, wird im Abschnitt 7.6 beschrieben.

7.5 BOBs programmieren

Einen BOB zu programmieren erfordert ziemlich viel Aufwand. Denn schließlich müssen bis zu drei Schritte hintereinander ausgeführt werden.

1. Hintergrund retten
2. Alten Hintergrund zurückschreiben
3. BOB in Hintergrund schreiben

Am besten legt man sich für jedes Objekt eine Tabelle (Struktur) an, in der alle notwendigen Parameter für die Darstellung eines BOBs enthalten sind. Diese Parameter werden dann von einer Unterroutine verwaltet. Sie berechnet dazu alle notwendigen Größen und trägt sie in die dazugehörigen Blitterregister ein. Das ist wohl die einfachste und schnellste Methode, ein Blitterobjekt auf dem Bildschirm erscheinen zu lassen. Möchte man jetzt zum Beispiel das Objekt bewegen, so wird einfach die Positionsvariable in der Tabelle geändert und die Unterroutine aufgerufen. Schon erscheint das Objekt an der neuen Position. Es ist allerdings auch möglich, die Blitterregister immer zum jeweiligen Augenblick an Ort und Stelle zu berechnen und einzutragen. Diese Methode ist aber sehr speicherraubend, erfordert sehr gute Kenntnisse der Blitterhardware und ist zudem umständlich. Trotz dieser Nachteile wird in den Computerzeitschriften meistens diese Methode aufgegriffen, um in irgendwelchen Grafikkursen einen BOB zu programmieren.

Wir werden unsere BOBs nach dem ersten Prinzip erzeugen. Dazu benötigen wir zwei Dinge. Einmal eine Tabelle, welche einen bestimmten Aufbau besitzt, um die Parameter zu speichern. Und natürlich die dazugehörige Unterroutine, welche die Parameter verwaltet. Das größere Problem von beiden ist die Unterroutine, daran scheitern auch die Computerzeitschriften. Denn diese Routine muß unabhängig alle Tabellen sofort verwalten können. Das ganze in einer gewaltigen Geschwindigkeit.

Für diesen Zweck habe ich eigens eine Routine entwickelt, mit der es kinderleicht ist, BOBs nach beschriebener Methode zu erzeugen. Diese Funktion, welche ich "PrintObjekt ()" genannt habe, kann sich wohl mit den schnellsten Routinen auf dem Amiga messen. Schneller als die GEL-Routinen (BOB- und Sprite-Routinen) des Betriebssystems ist sie allemal. Zudem ist die Routine relativ kurz. Die Funktion verwaltet beliebig viele Objekte, die jeweils eine maximale Größe von 1024 X 1024 Pixeln besitzen können. Sie schneidet auch die Objekte am Bildschirmrand entsprechend zurecht, damit beim Bewegen eines Objektes außerhalb der BitMap (Hintergrund) es nicht zu unschönen Effekten kommt, wie etwa Wiederhereinkommen auf der anderen Bildschirmseite oder sogar "Guru medetiert".

Die PrintObjekt () Funktion benötigt zwei Parameter. Erstens die Adresse der BitMap-Struktur, worauf das Objekt dargestellt werden soll. Diese wird in einer festen Variable gespeichert. Die Variable nennt sich "po_bitmap" und ist schon fest in der Funktion enthalten. Zweitens in dem Adressregister A0 die Anfangsadresse der Tabelle, welche alle nötigen Parameter enthält.

Die Tabelle, die alle nötigen Informationen enthält, besitzt den Namen "ObjektArgs" und ist folgendermaßen aufgebaut:

ObjektArgs-Struktur: (Länge = 32 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
00	Word	OldY	alte Y-Position (Intern für Routine)
02	Word	OldX	alte X-Position (Intern für Routine)
04	Byte	BOBOff	Wert/ Funktion: 0 Normal (führt Schritte 3/4/2 aus) 1 BOB wird ausgeschaltet (Schritt 3)

			2 Schreibe nur BOB-Daten in BitMap 3 Hintergrunddaten zurückschreiben 4 Hintergrunddaten lesen
05	Byte	Init	Variable für Routine
06	Word	YPos	Vertikale Position vom Objekt
08	Word	XPos	Horizontale Position vom Objekt
10	Word	Height	Höhe vom Objekt in Zeilen
12	Word	WordWidth	Breite vom Objekt in Words
14	Word	Depth	Anzahl Bitmaps (Planes) vom Objekt
16	Long	Image	Zeiger auf Grafikdaten vom Objekt
20	Long	ShadowMask	Zeiger auf Schattenmaske (Wichtig)
24	Long	SaveBuffer	Zeiger auf Hintergrundpuffer
28	Long	CollMask	Zeiger auf Collisionsmaske
32		END	Ende der ObjektArgs-Struktur

Erklärung:

OLdY,OldX (Offset 00/02):

Alte Koordinaten des Objekts. Wird von der Routine selbst verwaltet, darf nicht von Ihnen gesetzt werden. Diese Koordinaten sind für das richtige Restaurieren des Hintergrundes erforderlich. Sonst wüßte die Routine ja nicht, an welcher Stelle der alte Hintergrund zurückgeschrieben werden soll.

BOBOff (Offset 4):

Mit diesem Byte kann bestimmt werden, welcher Schritt ausgeführt werden soll. Dadurch ist man in der Lage, mehrere Prinzipien des DoubleBufferings durchzuführen (Kapitel 7.10). Will man jetzt nur das Objekt in den Hintergrund schreiben, ohne den Hintergrund zu retten, muß dieses Byte den Wert 2 enthalten.

Init (Offset 5):

Dieses Byte wird intern von der Routine verwaltet. Es muß beim ersten Aufruf der Routine auf Null gesetzt werden und darf danach nicht mehr geändert werden.

YPos,XPos (Offset 06/08):

Diese Words (16 Bits) enthalten die aktuelle Position des Objekts. Durch einfaches Ändern dieser Variablen. kann das Objekt bewegt werden.

Height (Offset 10):

Enthält die Anzahl Zeilen, die das Objekt groß ist. Das Objekt darf maximal 1024 Zeilen hoch sein.

WordWidth (Offset 12):

Hier wird die Breite des Objekts in Words (= 2 Bytes oder 16 Bits) eingetragen. In Words, weil der Blitter nur wordgroße Puffer bearbeiten kann. Achtung! - Das letzte Word, also die letzten 16 Bits einer Grafikzeile vom Objekt, muß immer Null sein. Sie müssen dazu das Objekt nur ein Word breiter machen, als es eigentlich ist. Durch diesen kleinen Trick ist man in der Lage, mit dem Blitter das Objekt an eine beliebige Position zu kopieren.

Depth (Offset 14):

Hier wird die Anzahl der Bitmaps gespeichert, die das Objekt besitzt. Also die Tiefe bzw. die Anzahl der Farben, die das Objekt maximal besitzen kann. Die Anzahl, die hier eingetragen wird, darf nicht größer sein, als die Anzahl in der BitMap-Struktur.

Image (Offset 16):

Hier wird die Anfangsadresse der ersten BitMap vom Objekt eingetragen. Dabei ist zu beachten, daß die BitMaps hintereinander liegen. Also erst BitMap 1, dann BitMap 2, usw., so viele, wie in der Variablen Depth angegeben sind.

ShadowMask (Offset 20):

Hier wird die Anfangsadresse auf einen Puffer eingetragen, der die Schattenmaske vom Objekt enthält. Eine Schattenmaske ist ein logisch ODER aller Bitmaps (Image) vom Objekt. Diese Schattenmaske ist notwendig, damit der Blitter weiß, ob jetzt ein Punkt gelöscht oder gesetzt werden soll. Denn es sollen ja nur die Punkte gesetzt werden, die auch in den BitMaps (Image) des Objekts gesetzt sind, alle anderen werden gelöscht. Für das Erzeugen der Schattenmaske wird im nächsten Abschnitt eine Routine vorgestellt. Die Größe des Puffers berechnet sich wie folgt: ShadowMaskSize = Height X WordWidth X 2 (In Bytes)

SaveBuffer (Offset 24):

Muß die Anfangsadresse des Puffers enthalten, worin der Hintergrund zwischengespeichert werden soll. Dieser Puffer ist nur erforderlich, wenn BOBOff die Werte 0, 3 oder 4 enthält. Die Puffergröße berechnet sich nach folgender Formel:

SaveBufferSize = Height x WordWidth x 2 x Screentiefe (Depth)

CollMask (Offset 28):

Enthält die Anfangsadresse der Collisionsmaske. Normalerweise besitzt sie den selben Aufbau, wie der ShadowMask-Puffer. Denn nur wo in der Collisionsmaske ein Punkt gesetzt ist, wird eine Collision zum Test herangezogen. Am besten sie setzen diesen Wert auf denselben vom ShadowMask. Außerdem spart das auch noch Speicherplatz.

Hier nocheinmal eine Kurzbeschreibung der Funktion "PrintObjekt ()":

Routine: PrintObjekt (A0,po_bitmap) (ObjektArgs,BitMap)
Parameter: A0 = Adresse der ObjektArgs-Struktur, welche alle nötigen Parameter enthält.
po_bitmap = Ist eine feste Variable im Listing, welches noch folgt, das die Adresse der BitMap-Struktur enthalten muß, in der das Objekt dargestellt werden soll.
Achtung!- Das letzte Word einer Grafikzeile muß immer Null sein.
Achtung!- Wenn man einen Hintergrund zurückschreibt, ohne vorher einen gerettet zu haben, wird die Hintergrundgrafik trotzdem nicht zerstört.

Jetzt endlich kommen wir zum eigentlichen Listing. Dieses ist so gut dukomentiert, daß eine ausführliche Beschreibung nicht erforderlich ist. Die Beschreibung ist ja auch mehr für die Profis unter Ihnen gedacht.

```
;--- Ein Objekt auf Bildschirm printen                    ---
;--- A0 = Objektargs                                      ---
;--- po_bitmap = Zeiger auf Bitmap-Struktur               ---
;--- Objekte nicht höher und/oder breiter als Bitmap      ---
;--- Diese Routine kann PC-Relative assembliert werden    ---
PrintObjekt:
  move.l po_bitmap(pc),a1
  move.l #$dff000,a2          ; Chip-Basisadresse $DFF000
  move.w (a1),d6
  lsr.w #1,d6                 ; D6 = Screenbreite in Words
  move.w 2(a1),d7             ; D7 = Screenhöhe
;
  move.w 6(a0),d0             ; Ypos nach D0
  cmp.w 2(a1),d0              ; If YPos >= Screenhöhe then
                                Ende
  bge po_clipping
  move.w 10(a0),d1            ; Höhe des Objekts nach D1
  beq.w d1                    ; negativ machen
  cmp.w d1,d0                 ; If YPos <= D1 then Ende
```

```
  ble po_clipping
  move.w 8(a0),d0          ; X-Pos. nach D0
  move.w (a1),d1           ; Zeilenbreite in Bytes
  lsl.w #3,d1              ; mal 8 = Zeilenbreite in Pixel
  cmp.w d1,d0              ; If XPos >= Zeilenbreite then
                             Ende
  bge po_clipping
  move.w 12(a0),d1         ; Objektbreite nach D1
  sub.w #1,d1              ; ein Word abziehen
  lsl.w #4,d1              ; mal 16 = Breite in Pixel
  neg.w d1                 ; Objektbreite jetzt negativ
  cmp.w d1,d0              ; If XPos <= D1 then Ende
  ble po_clipping
;
  cmp.b #1,4(a0)           ; BOBOff ?
  bne po_bobon
po_bob_aus:
  bsr po_writehintergrund
  clr.b 5(a0)              ; BOB nicht mehr init.
  rts
po_bobon:
  tst.b 4(a0)              ; Objekt normal anschalten ?
  bne po_bobon_xa
  bsr po_writehintergrund
  bsr po_readhintergrund   ; Hintergrund speichern
  bsr po_writeobjekt       ; Objekt in Hintergrund prin-
                             ten
  rts                      ; Ende
po_bobon_xa:
  cmp.b #2,4(a0)           ; Write BOB only
  bne po_bobon_xb
  bsr po_writeobjekt       ; Hintergrund zurückschrei-
                             ben
  rts
po_bobon_xb:
  cmp.b #3,4(a0)           ; Write Hintergrund only
  bne po_bobon_xc
  bsr po_writehintergrund
```

```
  rts
po_bobon_xc:
  cmp.b #4,4(a0)                ; Read Hintergrund only
  bne po_bobon_end
  bsr po_readhintergrund
po_bobon_end:
  rts
po_clipping:
  tst.b 4(a0)                   ; Objekt normal anschalten ?
  beq po_bob_aus
  cmp.b #1,4(a0)
  beq po_bob_aus
  cmp.b #3,4(a0)
  beq po_bob_aus
  rts

;--- Masken/Offset/Blittersize berechnen                    ---
;--- D0 = Y, D1 = X Position                                ---
;--- Rückgabe: D0 = Blitterhöhe, D1 = Blitterbreite         ---
;--- D2 = PositionOffset, D5 = Shiftwert                    ---
;--- D4 = muß zum Bitmapoffset addiert werden               ---
;
po_parameter:
  tst.w d0                      ; Y positiv ?
  bpl po_para_1                 ; wenn ja, dann verzweigen
  move.w 12(a0),d2              ; Breite Objekt Words
  lsl.w #1,d2                   ; in Bytes
  move.w d0,d3
  neg.w d3
  mulu d3,d2
  add.w 10(a0),d0               ; plus höhe Objekt
  bra po_para_x
po_para_1:
  move.w d7,d2                  ; Screenhöhe
  sub.w d0,d2                   ; minus y-pos
  move.w d2,d0                  ; D0 = ergebnis
  clr.l d2
  cmp.w 10(a0),d0               ; minus höhe
```

```
  bmi po_para_x                ; negativ ?
  move.w 10(a0),d0             ; wenn positiv dann normale
                                 Height
po_para_x:                     ; D0 = Blitterhöhe , D2 = Y-
                                 Offset
  tst.w d1                     ; X positiv ?
  bpl po_para_2                ; wenn ja, dann verzweigen
  neg.w d1                     ; X-pos jetzt positiv
  move.w d1,d4                 ; x-pos nach D4
  lsr.w #4,d1                  ; durch 16 teilen
  move.w 12(a0),d5             ; Objektbreite nach D5
  sub.w d1,d5
  clr.l d3
  move.w d1,d3
  lsl.w #1,d3                  ; D3 = X-Offset
  add.l d3,d2                  ; D2 = Position-Offset
  move.w d5,d1                 ; D1 = Blitterbreite
  and.w #15,d4
  clr.w d5
  tst.w d4
  beq po_para_44
  move.w #16,d5
  sub.w d4,d5                  ; D5 = real Shiftwert
  subq.w #1,d4
  move.w #$ffff,d3
po_para_shift:
  lsr.w #1,d3
  dbra d4,po_para_shift
  move.w d3,$44(a2)            ; FirstMask
  move.w d3,$46(a2)
  cmp.w #1,d1
  beq po_para_7
  move.w #$ffff,$46(a2)        ; LastMask
po_para_7:
  move.l #2,d4                 ; Bitmapoffset -2 Bytes
  rts
po_para_2:                     ; X-Pos ist positiv
  move.w d1,d5                 ; X-pos nach D5
```

```
  and.w #15,d5                    ; D5 = Shiftwert
  lsr.w #4,d1                     ; X-Pos durch 16
  move.w d6,d4                    ; Screenbreite nach D4
  sub.w d1,d4
  move.w d4,d1                    ; D1 = Blitterbreite
  cmp.w 12(a0),d1                 ; Ergebnis - Objektbreite
  bmi po_para_3                   ; hiernach normal weiter
  move.w 12(a0),d1                ; Blitterbreite
po_para_44:
  move.l #-1,$44(a2)              ; First/LastMask
  clr.l d4                        ; Bitmapoffset + 0 Bytes
  rts
po_para_3:
  tst.w d5
  beq po_para_44
  move.w d5,d3
  subq.w #1,d3
  move.w #$ffff,d4
po_para_shifta:
  lsl.w #1,d4
  dbra d3,po_para_shifta
  move.w d4,$44(a2)               ; FirstMask
  move.w d4,$46(a2)
  cmp.w #1,d1                     ; Blitterbreite = 1
  beq po_para_4
  move.w #$ffff,$44(a2)           ; LastMask
po_para_4:
  clr.l d4                        ; Bitmapoffset + 0 Bytes
  rts

;--- Hintergrund wieder printen ---
po_writehintergrund:
  tst.b 5(a0)                     ; wurde ein Hintergrund
                                    schon gelesen ?
  bne po_writehg_x
  rts
po_writehg_x:
  move.w (a0),d0                  ; OldY
```

```
  move.w 2(a0),d1                ; OldX
  bsr po_parameter
  move.l #-1,$44(a2)             ; First/LastMask
  move.l #$09f00000,$40(a2)      ; BLTCON0/1
  move.l 24(a0),a3               ; A3 = real Quelle A
  add.l d2,a3
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)              ; ZModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)              ; AModulo
  clr.l d3
  tst.w 2(a0)
  bmi po_writehg_nox
  move.w 2(a0),d3                ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                    ; X-Offset
po_writehg_nox:
  tst.w (a0)
  bmi po_writehg_noy
  move.w (a0),d4
  mulu d6,d4
  lsl.l #1,d4                    ; Y-Offset
  add.l d4,d3                    ; D3 = Bitmapoffset
po_writehg_noy:
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                    ; D4 = Map-Size vom Objekt
  clr.w d5
  move.b 5(a1),d5
  subq.w #1,d5                   ; D5 = Loop-zähler
  and.w #$3ff,d0                 ; Blittersize errechnen
  lsl.w #6,d0                    ; D1 = breite in Words
  and.w #$3f,d1                  ; D0 = höhe in Pixel
```

```
  add.w d0,d1                    ; D1 ist jetzt Blittersize
po_writehg_loop:
  move.l a3,$50(a2)              ; Quelle A
  move.l (a4)+,a5
  add.l d3,a5
  move.l a5,$54(a2)              ; Ziel D
  move.w d1,$58(a2)              ; Blitter starten
po_writehg_wait:
  btst #14,$2(a2)                ; Bit BBusy testen
  bne po_writehg_wait            ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_writehg_loop
  rts

;--- Hintergrund speichern ---
po_readhintergrund:
  move.b #1,5(a0)                ; Init = 1, Hintergrund schon
                                   mal gelesen
  move.w 6(a0),0(a0)             ; Ypos nach OldYpos
  move.w 8(a0),2(a0)             ; Xpos nach OldXpos
  move.w (a0),d0                 ; Y
  move.w 2(a0),d1                ; X
  bsr po_parameter
  move.l #-1,$44(a2)             ; First/LastMask
  move.l #$09f00000,$40(a2)      ; BLTCON0/1
  move.l 24(a0),a3               ; A3 = Ziel D
  add.l d2,a3
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)              ; AModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)              ; ZModulo
  clr.l d3
  tst.w 2(a0)
```

```
  bmi po_readhg_nox
  move.w 2(a0),d3              ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                  ; X-Offset
po_readhg_nox:
  tst.w (a0)
  bmi po_readhg_noy
  move.w (a0),d4
  mulu d6,d4
  lsl.l #1,d4                  ; Y-Offset
  add.l d4,d3                  ; D3 = Bitmapoffset
po_readhg_noy:
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                  ; D4 = Map-Size vom Objekt
  clr.w d5
  move.b 5(a1),d5
  subq.w #1,d5                 ; D5 = Loop-zähler
  and.w #$3ff,d0               ; Blittersize errechnen
  lsl.w #6,d0                  ; D1 = Breite in Words
  and.w #$3f,d1                ; D0 = Höhe in Pixel
  add.w d0,d1                  ; D1 ist jetzt Blittersize
po_readhg_loop:
  move.l a3,$54(a2)            ; Ziel D
  move.l (a4)+,a5
  add.l d3,a5
  move.l a5,$50(a2)            ; Quelle A
  move.w d1,$58(a2)            ; Blitter starten
po_readhg_wait:
  btst #14,$2(a2)              ; Bit BBusy testen
  bne po_readhg_wait           ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_readhg_loop
  rts

;--- Objekt Daten in Bitmap kopieren ---
po_writeobjekt:
  move.w 6(a0),d0              ; Y
```

```
  move.w 8(a0),d1                ; X
  bsr po_parameter
  lsl.w #8,d5
  lsl.w #4,d5                    ; korrekter Shiftwert
  move.w d5,$42(a2)              ; BLTCON1
  add.w #$0fca,d5
  movem.l d5,-(sp)
  move.w d5,$40(a2)              ; BLTCON0
  move.l 20(a0),a5
  add.l d2,a5                    ; A5 = Quelle A (real Shadow-
                                   Mask)
  move.l 16(a0),a3
  add.l d2,a3                    ; A3 = Quelle B (real Image)
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)              ; ZModulo
  move.w d5,$60(a2)              ; CModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)              ; AModulo
  move.w d5,$62(a2)              ; BModulo
  clr.l d3
  tst.w 8(a0)
  bmi po_writeo_nox
  move.w 8(a0),d3                ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                    ; X-Offset
po_writeo_nox:
  tst.w 6(a0)
  bmi po_writeo_noy
  move.w 6(a0),d5
  mulu d6,d5
  lsl.l #1,d5                    ; Y-Offset
  add.l d5,d3
po_writeo_noy:
```

```
  sub.l d4,d3                ; D3 = Bitmapoffset
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                ; D4 = Map-Size vom Objekt
  move.w 14(a0),d5
  subq.w #1,d5               ; D5 = Loop-zähler
  and.w #$3ff,d0
  and.w #$3f,d1
  lsl.w #6,d0                ; D1 = breite in Words
  add.w d0,d1                ; D1 ist jetzt Blittersize
po_writeo_loop:
  move.l a5,$50(a2)          ; Quelle A (ShadowMask)
  move.l a3,$4c(a2)          ; Quelle B (Image)
  move.l (a4)+,a6
  add.l d3,a6
  move.l a6,$48(a2)          ; Quelle C (BitMap)
  move.l a6,$54(a2)          ; Ziel D (BitMap)
  move.w d1,$58(a2)          ; Blitter starten
po_writeo_wait:
  btst #14,$2(a2)            ; Bit BBusy testen
  bne po_writeo_wait         ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_writeo_loop
  movem.l (sp)+,d5           ; BLTCON0 wert holen
  move.w 14(a0),d0           ; Depth BOB nach D0
  cmp.b 5(a1),d0             ; = Anzahl Planes Screen ?
  beq po_writeo_end          ; Wenn ja, dann Ende
  sub.b 5(a1),d0             ; Planes-Anzahl abziehen
  neg.b d0                   ; Positiv machen
  subq.w #1,d0
  sub.w #$0400,d5            ; DMA-Kanal B = aus
  move.w d5,$40(a2)          ; BLTCON0
  clr.w $42(a2)              ; No Shift B
  clr.w $72(a2)              ; Clear Datenregister B (Figur)

po_writeo_loop2:
  move.l a5,$50(a2)          ; Quelle A (ShadowMask)
  move.l (a4)+,a6
```

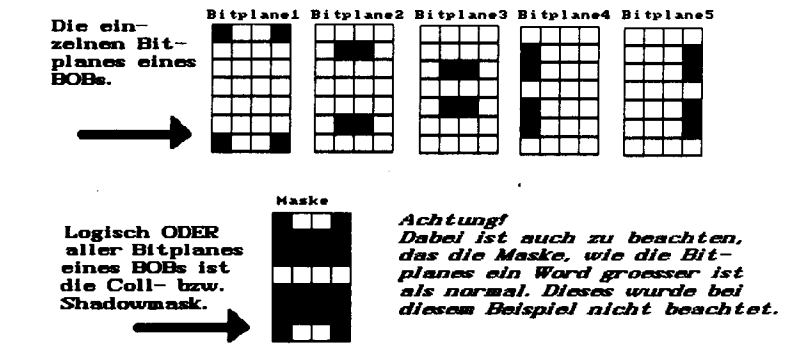
```
  add.l d3,a6
  move.l a6,$48(a2)          ; Quelle C (BitMap)
  move.l a6,$54(a2)          ; Ziel  D (BitMap)
  move.w d1,$58(a2)          ; Blitter starten
po_writeo_wait2:
  btst #14,$2(a2)            ; Bit BBusy testen
  bne po_writeo_wait2        ; wenn Null, dann Blitterende
  dbra d0,po_writeo_loop2

po_writeo_end:
  rts
po_bitmap: dc.l 0
```

7.6 Masken und Grafikpuffer

Wie aus der ObjektArgs-Struktur zu ersehen ist, sind einige Puffer für die Verwaltung nötig. Wie diese berechnet werden, wurde ja bereits schon beschrieben.

Man könnte jetzt natürlich umständlich für jedes Objekt die Puffergrößen berechnen. Das wäre jedoch viel zu umständlich und speicherraubend. Einfacher geht es, wenn eine Routine universell den Puffer für alle Objekte berechnen könnte. Man übergibt jediglich die Adresse der ObjektArgs-Struktur und die Funktion errechnet sich die Puffergröße über die entsprechenden Parameter selbst.

Es müssen also zwei Funktionen geschrieben werden, die einmal den SaveBuffer-Puffer berechnen und einmal den ShadowMask-Puffer mit der dazugehörigen Maske. Den CollMask-Puffer brauchen wir nicht zusätzlich zu berechnen, denn dieser sieht ja genauso aus, wie der ShadowMask-Puffer. Es sei denn, sie wollen für jedes Objekt eine eigene Collisionsmaske entwerfen.



Diese Funktionen sind bis auf das Erstellen der Schattenmaske ziemlich leicht zu programmieren. Wie schon erwähnt, handelt es sich dabei um ein logisches ODER aller Bitmaps vom Objekt. Das Einfachste wäre wohl mit dem Maschinensprachebefehl OR alle Bitmaps miteinander zu verknüpfen und das Ergebnis in den ShadowMask-Puffer abzulegen. Diese Methode funktioniert zwar, sie ist aber nicht gerade sehr schnell. Es gibt noch eine zweite Methode - sie werden es sich sicherlich schon denken können - mit dem Blitter. Damit sie sich nicht den Kopf darüber zerbrechen müssen, stelle ich ihnen zwei Funktionen vor, die das Erstellen der gewünschten Puffer übernehmen. Die eine, welche sich InitMask() nennt, benötigt zwei Parameter: Im Adressregister A0 die Adresse der Objekt Args-Struktur und im Datenregister D0 eine entsprechende Bedingung. Diese gibt an, ob vorher Speicher reserviert werden soll, und (oder) der CollMask-Puffer gleich dem Shadow-Mask -Puffer sein soll.

Die andere nennt sich GetSaveBuffer() und benötigt auch zwei Parameter. Ihr muß im Adressregister A0 die Adresse der Objekt-Args-Struktur und im Adressregister A1 die Adresse der BitMap-Struktur übergeben werden. Diese Funktion reserviert den Save-Buffer-Puffer für die Hintergrundspeicherung und trägt die Anfangsadresse des Puffers in die ObjektArgs-Struktur ein.

Routine: InitMask (A0,D0) (ObjektArgs,Bedingung)
Parameter: A0 = Adresse der ObjektArgs-Struktur
D0 = 0 - ShadowMask-Puffer wird nicht reserviert, sondern muß vorher schon reserviert worden sein. Es wird nur die Schattenmaske erzeugt und in diesem Puffer abgelegt.
D0 = 1 - ShadowMask-Puffer wird automatisch reserviert und Schattenmaske dort abgelegt.
D0 = 2 - Genau wie "1", nur wird zusätzlich der Coll-Mask-Zeiger gleich dem ShadowMask-Zeiger gesetzt.
Rückgabe: D0 = 0, alles OK!
D0 =-1, nicht genug Speicher vorhanden.
Erklärung: Initialisiert den ShadowMask-Puffer mit der entsprechenden Schattenmaske.

Routine: GetSaveBuffer (A0,A1) (ObjektArgs,BitMap)
Parameter: A0 = Anfangsadresse der ObjektArgs-Struktur
A1 = Anfangsadresse der BitMap-Struktur
Rückgabe: D0 = 0, dann ist alles in Ordnung!
D0 =-1, dann ist nicht genug Speicher vohanden!
Erklärung: Reserviert Speicher für die Hintergrundspeicherung und schreibt die Anfangsadresse in den SaveBuffer-Offset.

Mit den beiden Funktionen können wir den gewünschten Puffer jetzt erzeugen. Doch wie geben wir diesen wieder frei? Das erledigen die zwei folgenden Funktionen. Diese brauchen nicht näher erleutert zu werden, da sie sehr einfach aufgebaut sind.

Routine: FreeMask (A0) (ObjektArgs)
Parameter: A0 = Anfangsadresse der ObjektArgs-Struktur
Erklärung: Gibt den ShadowMask-Speicher, der mit InitMask () für die Schattenmaske reserviert worden ist, wieder zurück.

Routine: FreeSaveBuffer (A0,A1) (ObjektArgs,BitMap)
Parameter: A0 = Zeiger auf ObjektArgs-Struktur
A1 = Zeiger auf BitMap-Struktur
Erklärung: Gibt den Speicher, auf den SaveBuffer zeigt, wieder frei.

Als letztes nun die vier Listings. Diese sind ausführlich dokumentiert. Eine weitere Beschreibung ist deswegen nicht erforderlich.

```
;--- ShadowMask anlegen                    ---
;--- A0 = ObjektArgs, D0 = Bedingung       ---

InitMask:
  tst.w d0                          ; Muß ShadowMask-Puffer reserviert werden ?
  beq im_1
  movem.l d0,-(sp)
  move.w 12(a0),d0                  ; WordWidth
  lsl.w #1,d0                       ; mal 2 = Bytes
  mulu 10(a0),d0                    ; mal Höhe = ShadowMask-Size
  move.l #$10002,d1                 ; Chip und ClearMem
  move.l #$4,a6
  move.l (a6),a6
  movem.l a0,-(sp)
  jsr -198(a6)                      ; AllocMem
  movem.l (sp)+,a0
  tst.l d0
  bne im_1a
  movem.l (sp)+,d0
  move.l #-1,d0                     ; Error
  rts
im_1a:
  move.l d0,20(a0)                  ; ShadowMask-Adresse speichern
  movem.l (sp)+,d0
  cmp.w #2,d0
  bne im_1
```

```
  move.l 20(a0),28(a0)
im_1:
  move.l #$dff000,a6
  move.l #$0dfc0000,$40(a6)     ; BLTCON0: A + B = D
  move.l #-1,$44(a6)            ; First/Last Mask
  clr.l $62(a6)                 ; Modulowert von Quelle B
  clr.w $66(a6)                 ; Modulowert von Ziel  D
  move.w 12(a0),d5              ; Breite in Words
  move.w 10(a0),d6              ; Höhe in Pixel
  move.w d5,d0                  ; Breite nach D0
  lsl.w #1,d0                   ; mal 2 = Breite in Bytes
  mulu d6,d0                    ; D0 = ImageMapSize
  and.w #$3ff,d6                ; Blittersize errechnen
  lsl.w #6,d6                   ; D5 = Breite in Words
  and.w #$3f,d5                 ; D6 = Höhe in Pixel
  add.w d6,d5                   ; D5 ist jetzt Blittersize
  move.w 14(a0),d1              ; Anzahl Planes nach D1
  sub.w #1,d1                   ; minus 1, wegen DBra
  move.l 20(a0),a1              ; A1 = ShadowMask (Ziel D
                                  und Quelle B)
  move.l 16(a0),a2              ; A2 = Image-Zeiger
initmask_loop:
  move.l a2,$50(a6)             ; Anfangsadresse von Quelle
                                  A
  move.l a1,$54(a6)             ; Anfangsadresse von Ziel D
  move.l a1,$4c(a6)             ; Anfangsadresse von Quelle
                                  B = Ziel D
  move.w d5,$58(a6)             ; BLTSIZE und Blitteroperati-
                                  on starten
initmask_wait:
  btst #14,$2(a6)               ; Blitter fertig?
  bne initmask_wait
  add.l d0,a2                   ; nächste ImageMap
  dbra d1,initmask_loop
  clr.l d0                      ; No Errors
  rts
;--- Listingende ---
```

```
;---- Speicher für SaveBuffer reservieren      ---
;--- A0 = ObjektArgs; A1 = BitMap              ---

GetSaveBuffer:
  move.w 12(a0),d0              ; WordWidth
  lsl.w #1,d0                   ; mal 2 = Bytes
  mulu 10(a0),d0                ; mal Höhe  = MapSize
  move.b 5(a1),d1               ; Screen-Depth nach D1
  and.w #$00ff,d1               ; Hi-Byte löschen
  mulu d1,d0                    ; Screen-Depth mal MapSize
  move.l #$10002,d1             ; Chip + ClearMem
  move.l #$4,a6
  move.l (a6),a6
  movem.l a0,-(sp)
  jsr -198(a6)                  ; AllocMem
  movem.l (sp)+,a0
  tst.l d0
  bne gsb_1
  move.l #-1,d0                 ; Error
  rts
gsb_1:
  move.l d0,24(a0)              ; SaveBuffer-Zeiger init.
  clr.l d0
  rts
;--- Listingende ---

;---- ShadowMask-Puffer wieder freigeben       ---
;--- A0 = ObjektArgs                           ---
FreeMask:
  move.l 20(a0),a1              ; ShadowMask-Zeiger
  cmp.l #0,a1                   ; vorhanden ?
  bne fm_1
  rts
fm_1:
  clr.l 20(a0)                  ; ShadowMask-Zeiger löschen
  move.w 12(a0),d0              ; WordWidth
  tsl.w #1,d0                   ; mal 2 = ByteWidth
  mulu 10(a0),d0                ; mal  Höhe  =  ShadowMask-
```

```
                                    Size .
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                      ; FreeMem
  rts
;--- Listingende ---


;--- SaveBuffer-Speicher wieder freigeben          ---
;--- A0 = ObjektArgs; A1 = BitMap                  ---
FreeSaveBuffer:
  move.w 12(a0),d0                  ; WordWidth
  lsl.w #1,d0                       ; mal 2 = Bytes
  mulu 10(a0),d0                    ; mal Höhe = MapSize
  move.b 5(a1),d1                   ; Screen-Depth nach D1
  and.w #$00ff,d1                   ; Hi-Byte löschen
  mulu d1,d0                        ; Screen-Depth mal MapSize
  move.l 24(a0),a1
  cmp.l #0,a1
  bne fsb_1
  rts
fsb_1:
  clr.l 24(a0)                      ; SaveBuffer-Zeiger löschen
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                      ; FreeMem
  rts
;--- Listingende ---
```

7.7 BOB - Routinen

Ein Blitterobjekt (BOB) kann in mehreren Arten über den Bildschirm bewegt werden. Einmal ganz normal, über eine BitMap oder doppeltgepuffert über mindestens zwei BitMaps. Auf die letzte Methode wird im Kapitel 7.10 eingegangen. Wir werden hier auf das ganz einfache Bewegen eines Objekts über einer BitMap eingehen.

Dazu muß auf jedem Fall der Hintergrund jedesmal gerettet werden, ansonsten würden wir das Objekt wie einen Pinsel benutzen. Dazu müssen wir nach folgender Art vorgehen:

1. alten Hintergrund zurückschreiben (an alte Positionen)
2. Hintergrund ab den neuen Positionen retten
3. Objekt in den Hintergrund schreiben

Das sind immerhin drei Schritte, die auszuführen sind. Bei 12 Objekten ist das ein ganz schöner Aufwand, da wir ja alle drei Schritte jedesmal für jedes einzelne Objekt durchführen müssen. Damit dieser Aufwand so klein wie möglich gehalten wird, stelle ich ihnen einige Routinen vor, die jeweils für alle Objekte nur einmal aufgerufen werden. Dabei spielt es keine Rolle, wieviele Objekte dargestellt werden sollen.

Das Prinzip der Funktionen ist ziemlich einfach. In der Variablen BOBOFF (ObjektArgs-Struktur) wird ja festgelegt, welcher Schritt gerade ausgeführt werden soll. Wir legen uns jetzt einfach eine Tabelle an, in der alle Objekte enthalten sind und rufen zum Beispiel die Funktion auf, welche jetzt für alle Objekte den Hintergrund zurückschreibt, also BOBOFF auf den Wert 3 setzt. Nach genau diesem Prinzip laufen auch die anderen Funktionen ab.

Achtung! - Bei allen Funktionen, die gleich beschrieben werden, muß sich zusätzlich die Routine PrintObjekt () im Speicher befinden, denn die Funktionen greifen auf diese Routine zurück, um die Objekte darstellen zu können.

Die Funktionen benötigen nur einen Parameter. Nämlich in der Variablen printbob_tabelle die Adresse der BOB-Tabelle, in der alle Objekte enthalten sind. Diese Variable muß vier Bytes lang sein und von ihnen selbst angelegt werden. Zum Beispiel:

printbob_tabelle: dc.l 0

Die BOB-Tabelle ist folgendermaßen aufgebaut:

BOB-Tabelle: (Länge mindestens 6 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
00	Word	Count	Anzahl ObjektArgs-Strukturen
02	Long	ObjektArgs0	Zeiger auf ObjektArgs-Struktur von BOB 0
06	Long	ObjektArgs1	Zeiger auf ObjektArgs-Struktur von BOB 1

... so oft wie in Count angegeben ist.

In Count wird angegeben, wieviel ObjektArgs-Strukturen die BOB-Tabelle enthält. Ab Offset zwei stehen die Anfangsadressen der ObjektArgs-Strukturen. Dabei besitzt das zuletzt eingetragene Objekt die höchste Priorität, verdeckt alle dahinterliegenden Objekte. Nun zu den eigentlichen Funktionen. Sie werden hier in Kurzform beschrieben. Danach folgen die dazugehörigen Listings.

Routine: WriteBackgroundsOnly (printbob_tabelle) (BOB-Tabelle)
Parameter: printbob_tabelle = Adresse der BOB-Tabelle, in der alle BOBs enthalten sind.
Erklärung: Schreibt den Hintergrund von allen BOBs an den alten Positionen in die BitMap zurück.

Routine: ReadBackgroundsOnly (printbob_tabelle) (BOB-Tabelle)
Parameter: siehe WriteBackgroundsOnly ()
Erklärung: Kopiert den Hintergrund von allen BOBs ab den neuen Positionen aus der BitMap in den SaveBuffer-Speicher.

Routine: WriteBobsOnly (printbob_tabelle) (BOB-Tabelle)
Parameter: siehe WriteBackgroundsOnly ()
Erklärung: Kopiert die BOB-Imagedaten von allen BOBs in die Bit-Map (Hintergrund).

Hier die dazugehörigen Listings:

```
;--- Nur SaveBuffer-Daten in BitMap zurückkopieren          ---
;--- printbob_tabelle = BOB-Tabelle                         ---

WriteBackgroundsOnly:
  move.l printbob_tabelle(pc),a1
  move.w (a1)+,d0                ; Anzahl Einträge
  tst.w d0
  beq writebackgroundsonly_end
  clr.l d1
  move.w d0,d1
  lsl.w #2,d1                    ; Anzahl mal 4
  add.l d1,a1                    ; Zeiger auf letzten Eintrag
  subq.w #1,d0                   ; Anzahl minus 1
wbg_loop:
  move.l -(a1),a0                ; Zeiger auf ObjektArgs
  cmp.l #0,a0                    ; überhaupt vorhanden ?
  beq wbg_loop1
  move.b #3,4(a0)                ; Write Background only
  movem.l d0/a0-a1,-(sp)
  bsr printobjekt
  movem.l (sp)+,d0/a0-a1
wbg_loop1:
  dbra d0,wbg_loop
writebackgroundsonly_end:
  rts
;--- Listingende ---
```

```
;--- Nur Hintergrund von BitMap nach SaveBuffer kopieren ---
;--- printbob_tabelle = BOB-Tabelle                       ---

ReadBackgroundsOnly:
  move.l printbob_tabelle(pc),a1
  move.w (a1)+,d0                    ; Anzahl Einträge
  tst.w d0
  beq readbackgroundsonly_end
  clr.l d1
  move.w d0,d1
  lsl.w #2,d1                        ; Anzahl mal 4
  add.l d1,a1                        ; Zeiger auf letzten Eintrag
  subq.w #1,d0                       ; Anzahl minus 1
rb_loop:
  move.l -(a1),a0                    ; Zeiger auf ObjektArgs
  cmp.l #0,a0                        ; überhaupt vorhanden ?
  beq rb_loop1
  move.b #4,4(a0)                    ; Read Background only
  movem.l d0/a0-a1,-(sp)
  bsr printobjekt
  movem.l (sp)+,d0/a0-a1
rb_loop1:
  dbra d0,rb_loop
readbackgroundsonly_end:
  rts
;--- Listingende ---


;--- Nur BOB-Imagedaten in BitMap kopieren      ---
;--- printbob_tabelle = BOB-Tabelle             ---

WriteBobsOnly:
  move.l printbob_tabelle(pc),a1
  move.w (a1)+,d0                    ; Anzahl Einträge
  tst.w d0
  beq writebobsonly_end
  clr.l d1
  move.w d0,d1
```

```
  lsl.w #2,d1                ; Anzahl mal 4
  add.l d1,a1                ; Zeiger auf letzten Eintrag
  subq.w #1,d0               ; Anzahl minus 1
wb_loop:
  move.l -(a1),a0            ; Zeiger auf ObjektArgs
  cmp.l #0,a0                ; überhaupt vorhanden ?
  beq wb_loop1
  move.b #2,4(a0)            ; Write BOB only
  movem.l d0/a0-a1,-(sp)
  bsr printobjekt
  movem.l (sp)+,d0/a0-a1
wb_loop1:
  dbra d0,wb_loop
writebobsonly_end:
  rts
;--- Listingende ---

printbob_tabelle: dc.l 0     ; Variable für BOB-Tabelle
```

7.8 IFF-Brushes in BOBs umwandeln

Jetzt wissen wir, wie ein BOB aufgebaut ist und wie wir diesen darstellen können. Doch ist es ein ziemlich großer Aufwand, einen BOB zu zeichnen. Entweder legen wir die Grafikdaten direkt im Speicher ab, über umständliche "dc.b" -Assemblerbefehle oder wir besitzen einen BOB-Editor, mit dem wir unsere Grafik zeichnen und abspeichern können. Doch wer besitzt schon einen BOB-Editor? Die Editoren, die erhältlich sind, sind in der Regel nicht gerade sehr komfortabel bzw. leistungsfähig. Die meisten unter ihnen werden aber bestimmt ein Malprogramm ihr eigen nennen können, das bekannteste ist wohl Deluxe-Paint. Dieses enthält unter anderem einige Brush-Funktionen. Dafür werden Teile aus einer Grafik herausgeschnitten und können auf vielfältige Weise manipuliert und auf Diskette abgespeichert werden. Diese Brushes oder anders Pinsel sind nichts anderes als Blitterobjekte - kurz BOBs.

Um jetzt aber einen solchen Brush in sein Programm einbauen zu können, müßte man den Aufbau der Brushdatei kennen. Dieser ist jedoch relativ einfach, denn alle Brushes werden im sogenannten IFF-Standart abgespeichert. Dieser Standart ist auf dem Amiga sehr verbreitet. Er wird nicht nur für die Grafik, sondern auch für den Sound oder Text benutzt. Durch diesen Standart ist der Datenaustausch zwischen verschiedenen Programmen möglich geworden.

Auf den genauen Aufbau wollen wir hier nicht weiter eingehen. Ich werde ihnen aber ein Programm vorstellen, mit dem sie IFF-Brushes in den Speicher laden und als BOBs abspeichern können.

Die BOB-Datei, welche auf Diskette abgespeichert wird, gliedert sich in zwei Teile: In den Header und den eigentlichen Grafikteil. In dem Header, der die ersten 40 Bytes darstellt, stehen alle nötigen Parameter, die für das Darstellen des BOBs erforderlich sind. Danach folgen dann die eigentlichen Grafikdaten.

Aufbau der BOB-Headerdatei:

Offset	Typ	Bezeichnung	Beschreibung
00	Long	TBOB	Kennzeichnung der Headerdatei
04	Long	V1.0	Transformerversion
08	Long	Länge	Länge der gesammten BOB-Datei
12	Long	HEAD	4-Byte Kennzeichnung
16	Word	Höhe	Höhe des BOBs in Zeilen
18	Word	WordBreite	Breite des BOBs in Words (1 Word = 16 Bit)
20	Word	Breite	Breite des BOBs in Pixeln
22	Word	Tiefe	Anzahl Bitmaps des BOBs
24	Word	Frei	unwichtig
26	Word	YPos	Y-Position vom BOB
28	Word	XPos	X-Position vom BOB

30	Word	Frei	unwichtig
32	Long	BODY	Kennzeichnung der Imagedaten
36	Long	Grafiklänge	Länge der eigentlichen Grafikdaten
40	Byte		ab hier liegen die eigenlichen Grafikdaten

Beschreibung der Parameter:

TBOB (Offset 00):

Kennzeichnung des Brushformers. Die ersten 4 Bytes enthalten die vier Großbuchstaben TBOB. Für den Benutzer nicht von Bedeutung.

V1.0 (Offset 04):

Enthält die Versionsnummer des Brushformers, mit dem der BOB erstellt wurde. Auch wieder als vier Großbuchstaben.

Länge (Offset 08):

Enthält die gesamte Länge der BOB-Datei in Bytes.

HEAD (Offset 12):

Kennzeichnung, daß ab hier die eigentlichen Parameter folgen.

Höhe (Offset 16):

Hier steht die Höhe des Objekts in Zeilen.

WordBreite (Offset 18):

Enthält die Breite des Objekts in Words.

Breite (Offset 20):

Breite des Objekts in Pixeln (unwichtig)

Tiefe (Offset 22):

Enthält die Anzahl BitMaps, die das Objekt groß ist, also aus wieviel Farben das Objekt maximal besteht.

YPos,XPos (Offset 26/28):

Enthält die Koordinaten des Objekts. In dieser Brushfomer-Version besitzten diese Variablen immer den Wert Null.

BODY (Offset 32):

Kennzeichnung, daß ab hier die eigentlichen Grafikinformationen folgen. Wird als vier Großbuchstaben abgelegt.

Grafiklänge (Offset 36):

Hier steht die Länge der eigentlichen Grafikdaten, also die gesamten Imagedaten zusammen.

Imagedaten (Offset 40):

Ab dem 40. Byte liegen die eigentlichen Imagedaten. Erst die von BitMap 1, danach die von BitMap 2 usw. So viele, wie in Tiefe angegeben ist.

Das folgende Listing des Brushformers V1.0 ist sehr gut dokumentiert und sie finden es auch auf der beigefügten Diskette.

```
;--- Programmname = Brushformer                    ---
;--- wandelt ein Brush in einen BOB um             ---
;
;--- Start von cli oder wb ---
  move.l 4,a6
  move.l #0,a1
  jsr -294(a6)                    ; FindTask ()
  move.l d0,a4
  tst.l $ac(a4)
  bne start                       ; Cli start
  move.l 4,a6
  lea $5c(a4),a0
  jsr -384(a6)                    ; WaitPort ()
  jsr -372(a6)                    ; Message abholen, weil Wor-
                                    bench start

start:
  bsr openlibrary
  bsr openworkscreen
  bsr openworkwindow
  bsr initmenü

main:
  bsr message                     ; Message abholen (Menüs
                                    abfragen)
  cmp.l #"NEIN",d5                ; Menüpunkt angewählt?
  beq main                        ; wenn nicht, dann weiter
                                    warten
  bra auswerten                   ; wenn ja, dann Menünummer
                                    verarbeiten

EXIT:
  bsr closeworkwindow
  bsr closeworkscreen
  bsr closelibrary
  move.l bob_planes_base,a1       ; Befindet sich ein BOB im
                                    Speicher
  cmp.l #0,a1                     ; wenn nicht, dann weiter
```

```
  beq exit_1
  move.l bob_planes_size,d0        ; sonst Speicher wieder
  move.l 4,a6                      ; freigeben
  jsr -210(a6)                     ; FreeMem ()
exit_1:
  rts                              ; PRG.-Ende

;--- Arbeitsfenster öffnen ---
openworkwindow:
  move.l workscreenhd,workshd      ; Screenadresse
  move.l #workwindow,a0            ; Window-Struktur
  move.l intbase,a6                ; intuition.basis
  jsr -204(a6)                     ; OpenWindow ()
  move.l d0,workwindowhd           ; Windowadresse
  rts

;--- Arbeitswindow schließen ---
closeworkwindow:
  move.l workwindowhd,a0           ; Windowadresse
  move.l intbase,a6                ; intuition.basis
  jsr -72(a6)                      ; CloseWindow ()
  rts

;--- Arbeitsscreen öffnen ---
openworkscreen:
  move.l #workscreen,a0            ; Screen-Struktur
  move.l intbase,a6                ; intition.basis
  jsr -198(a6)                     ; OpenScreen ()
  move.l d0,workscreenhd           ; Screenadresse
  rts

;--- Arbeitsscreen schließen ---
closeworkscreen:
  move.l workscreenhd,a0           ; Screenadresse
  move.l intbase,a6                ; intuition.basis
  jsr -66(a6)                      ; CloseScreen ()
  rts
```

```
;--- Bibliotheken öffnen ---
openlibrary:
  move.l #intname,a1          ; Intuitionsbiblitothek
  bsr openlib                 ; öffnen
  move.l d0,intbase           ; Intuitionsbasisadresse
  move.l #gfxname,a1          ; Graphicsbibliothek
  bsr openlib                 ; öffnen
  move.l d0,gfxbase           ; Graphicsbasisadresse
  move.l #dosname,a1          ; Dosbibliothek
  bsr openlib                 ; öffnen
  move.l d0,dosbase           ; Dosbasisadresse
  rts
openlib:
  clr.l d0                    ; Version = 0
  move.l 4,a6                 ; Execbasis
  jsr -552(a6)                ; OpenLibrary ()
  rts

;---Bibliotheken schließen ---
closelibrary:
  move.l dosbase,a1           ; Dosbasisadresse
  move.l 4,a6                 ; Execbasisadresse
  jsr -414(a6)                ; CloseLibrary ()
  move.l intbase,a1           ; Intuitionsbasisadresse
  move.l 4,a6                 ; Execbasisadresse
  jsr -414(a6)                ; CloseLibrary ()
  move.l gfxbase,a1           ; Graphicsbasisadresse
  move.l 4,a6                 ; Execbasisadresse
  jsr -414(a6)                ; CloseLibrary ()
  rts

;--- Menü aktivieren ---
initmenü:
  move.l intbase,a6           ; Intuitionsbasis
  move.l workwindowhd,a0      ; Arbeitswindowadresse
  move.l #menü0,a1            ; Menüadresse
  jsr -264(a6)                ; SetMenüStrip ()
  rts
```

```
;--- Message abholen (Menüs abfragen) ---
message:
  move.l 4,a6                    ; Execbasisadresse
  move.l workwindowhd,a0         ; Arbeitswindowadresse
  move.l 86(a0),a0               ; UserMessagePort-Adresse
  jsr -372(a6)                   ; GetMsg ()
  cmp.l #0,d0                    ; Message vorhanden ?
  beq messageend                 ; keine Message - Ende
  move.l d0,a0
  move.l $14(a0),d6              ; IDCMP-Flags nach D6
messagemenü:
  cmp.l #$100,d6                 ; Wurde ein Menüpunkt ange-
                                   wählt?
  bne messageend                 ; Wenn nicht - Ende
  move.w $18(a0),d7              ; Menünummer holen
  cmp.w #$ffff,d7                ; kein Menü ?
  beq messageend                 ; dann Ende
;---- Menünummer auswerten ----
  move.w d7,d6
  lsr.w #8,d6
  lsr.w #3,d6                    ; d6 = untermenüpunktnum-
                                   mer
  move.w d7,d5
  lsl.w #8,d5
  lsl.w #3,d5
  lsr.w #8,d5
  lsr.w #3,d5                    ; d5 = menünummer oder
                                   "NEIN"
  lsl.w #5,d7
  lsr.w #8,d7
  lsr.w #2,d7                    ; d7 = menüpunktnummer
  rts
messageend:
  move.l #"NEIN",d5
  rts
```

```
;--- Menüpunkt ausführen ---
auswerten:
  clr.w d0
  auswert0:
  cmp.w #0,d5                ; Menü 0?
  beq auswert1
  bra main                   ; wenn nicht, dann Haupt-
                               schleife

auswert1:
  move.l #sprungtabelle1,a0  ; Menü 0
auswertung:
  cmp.w d0,d7                ; wurde ein Menü angewählt ?
  beq routine                ; ja, dann aufrufen
  cmp.w #6,d0                ; max. Anzahl Menüpunkte =
                               6
  beq main                   ; zurück zur Hauptschleife
  add.w #1,d0                ; nächsten Menüpunkt
  add.l #4,a0                ; nächste Menüadresse in Ta-
                               belle
  bra auswertung             ; weiter abfragen
routine:
  move.l (a0),jump+2         ; Menüfunktion ausführen
jump:
  jmp jump                   ; Auf keinen Fall eine 0 ein-
                               setzen
;                            ; Das ist der Unterschied zu
                               SEKA!
sprungtabelle1:
dc.l LOAD_BRUSH,SAVE_BOB,EXIT


;---- IFF-Brush in Speicher laden und in BOB umwandeln ---
load_brush:
  bsr loadbrush
  bra main
;
```

```
loadbrush:
  move.l bob_planes_base,a1      ; Befindet sich ein BOB im
                                   Speicher
  cmp.l #0,a1                    ; wenn nicht, dann weiter
  beq loadbrush_1
  move.l bob_planes_size,d0      ; sonst Speicher wieder
  move.l 4,a6                    ; freigeben
  jsr -210(a6)                   ; FreeMem ()
  clr.l bob_planes_base
loadbrush_1:
;--- IFF-Header in Speicher laden ---
  move.l #1005,d2                ; Mode = OLD
  move.l #filename,d1            ; Filename
  move.l dosbase,a6
  jsr -30(a6)                    ; OPEN ()
  move.l d0,filehd               ; Fileadresse speichern
  tst.l d0                       ; File auf Disk ?
  bne ok1                        ; wenn ja, dann ok1
  lea text2,a0                   ; Textadresse
  bsr printtext                  ; und ausgebn (Error)
  rts
ok1:
  move.l filehd,d1               ; Fileadresse
  move.l #puffer,d2              ; Puffer für
  move.l #12,d3                  ; ersten 12 Bytes
  move.l dosbase,a6
  jsr -42(a6)                    ; READ ()
  move.l filehd,d1               ; File wieder schließen
  move.l dosbase,a6
  jsr -36(a6)                    ; CLOSE ()
;--- ab hier Diskheader auswerten ---
  lea puffer,a0
  cmp.l #"FORM",(a0)             ; liegt IFF-Datei vor ?
  beq ok2                        ; wenn ja, dann ok2
  lea text3,a0                   ; textadresse
  bsr printtext                  ; und ausgeben (Error)
  rts
```

```
ok2:
  cmp.l #"ILBM",8(a0)          ; liegt ILBM-Format vor ?
  beq ok3                      ; wenn ja, dann ok3
  lea text4,a0                 ; textadresse
  bsr printtext                ; und ausgeben (Error)
  rts                          ; Programmende
ok3:
;--- Speicher für Brush reservieren ---
  move.l 4(a0),d0              ; Dateigröße
  move.l #$10002,d1            ; Chip- und Free-Memory
  move.l 4,a6
  jsr -198(a6)                 ; AllocMem ()
  move.l d0,brush_base         ; Adresse speichern
  tst.l d0                     ; konnte Speicher reserviert
                                 werden ?
  bne ok4                      ; wenn ja, dann ok4
  lea text5,a0                 ; sonst Errormeldung
  bsr printtext                ; ausgeben
  rts
ok4:
;--- Brush in Speicher laden ---
  move.l #1005,d2              ; Modus = Alt
  move.l #filename,d1          ; Filename
  move.l dosbase,a6
  jsr -30(a6)                  ; OPEN ()
  move.l d0,filehd             ; Fileadresse
  move.l d0,d1
  move.l brush_base,d2         ; Datenpuffer
  move.l puffer+4,d3           ; Datenlänge
  move.l dosbase,a6
  jsr -42(a6)                  ; READ ()
  move.l filehd,d1             ; Fileadresse
  move.l dosbase,a6
  jsr -36(a6)                  ; CLOSE

;--- IFF-Brush-Struktur auswerten ---
  move.l #chunktabelle,a1      ; Brush-Chunkadressen su-
                                 chen
```

```
  move.l #chunkadresse,a2          ; dort werden Adressen ge-
                                     speichert
biffsearch0:
  move.l brush_base,a0             ; Brushanfang
biffsearch1:
  cmp.l #0,(a1)                    ; schon alle Chunks gefun-
                                     den?
  beq chunk_search_ende            ; wenn ja, dann Ende
  move.b (a0)+,iffchunk            ; sonst
  move.b (a0)+,iffchunk+1          ; erstes
  move.b (a0)+,iffchunk+2          ; LongWord
  move.b (a0)+,iffchunk+3          ; lesen
  move.l iffchunk,d6               ; LongWord nach D6
  move.l (a1),d7                   ; mit Chunk aus
  cmp.l d7,d6                      ; Tabelle vergleichen
  beq biffsearch3                  ; wenn gleich, dann Adresse
                                     speichern
  sub.l #3,a0                      ; sonst Brushadresse minus 3
  bra biffsearch1                  ; und weiter suchen
biffsearch3:                       ; ab hier Chunkadresse spei-
                                     chern
  sub.l #4,a0                      ; Brushadresse minus 4
  move.l (a2),a4                   ; Puffer aus Tabelle holen
  move.l a0,(a4)                   ; und Chunkadresse spei-
                                     chern
  add.l #4,a1                      ; Chunktabelle plus 4
  add.l #4,a2                      ; Puffertabelle plus 4
  bra biffsearch0                  ; nächsten Chunk as Tabelle
                                     suchen
chunk_search_ende:
; --- BMHD-Chunk auswerten für Unpacker ---
  move.l bmhd_chunk,a0             ; BMHD-Chunkadresse nach
                                     A0
  move.b 18(a0),packerbyte         ; Compression - Byte spei-
                                     chern
  move.w 10(a0),bob_höhe           ; Bobhöhe speichern
  clr.l d0
  move.w 8(a0),d0                  ; BOB-Breite nach D0
```

```
  divu #16,d0                ; durch 16 teilen
  swap d0                    ; HIGH- und LOW-Word vertauschen
  cmp.w #0,d0                ; Rest der Division = Null ?
  bne daw1                   ; wenn nicht, dann daw1
  swap d0                    ; sonst, Words wieder vertauschen
  bra daw2                   ; und daw2
daw1:
  swap d0                    ; Words wieder vertauschen
  add.w #1,d0                ; plus 1 Word (wegen Rest)
daw2:
  lsl.w #1,d0                ; Wordbreite mal 2 = Bytebreite
  move.w d0,bob_bytebreite   ; Bobbreite (in Bytes) speichern
  clr.w d0
  move.b 16(a0),d0           ; Depth vom Brush
  move.w d0,bob_planes       ; Anzahl Bob-Planes speichern

;--- Speicher fürs Entpacken reservieren ----
  move.w bob_planes,d0       ; Anzahl Planes
  mulu bob_bytebreite,d0     ; mal BOB-Breite
  mulu bob_höhe,d0           ; mal BOB-Höhe
  move.l d0,bob_planes_size  ; = BOB-Größe
  move.l #$10002,d1          ; CHIP- und FREE-Memory
  move.l 4,a6
  jsr -198(a6)               ; AllocMem ()
  move.l d0,bob_planes_base  ; Adresse speichern
  tst.l d0                   ; konnte Speicher reserviert werden ?
  bne bob_ok6                ; wenn ja, dann ok6
  lea text5,a0               ; sonst Errormeldung
  bsr printtext              ; ausgeben
  move.l brush_base,a1       ; Wurde Speicher für
  move.l puffer+4,d0         ; sonst Speicher wieder
  move.l 4,a6                ; freigeben
```

```
  jsr -210(a6)                    ; FreeMem ()
  rts

;--- Anfangsadresse der BitMaps errechnen und speichern ---
bob_ok6:
  move.l d0,a0                    ; Adresse der 1. Bitplane
  move.w bob_bytebreite,d0        ; BOB-Bytebreite nach D0
  mulu bob_höhe,d0                ; mal BOB-Höhe = Bitplane-
                                    größe
  move.l #bitmapadressen,a1       ; Puffer für Bitplane-Pointer
  move.w bob_planes,d1            ; Anzahl Planes
  sub.w #1,d1                     ; minus 1, wegen DBRA-Be-
                                    fehl
buploop:
  move.l a0,(a1)+                 ; Bitplaneadresse speichern
  add.l d0,a0                     ; plus eine Bitplanegröße
  dbra d1,buploop                 ; weiter speichern

;--- Brush entpacken ---
  bsr start_dekompression
;
;--- BOB-Header aufbauen ---
  lea bob_header,a0               ; Headeranfangsadresse
  move.l #"TBOB",(a0)             ; Bezeichnung
  move.l #"V1.0",4(a0)            ; Version
  move.l bob_planes_size,8(a0)    ; BOB-ImageSize
  add.l #40,8(a0)                 ; plus BOB-Header
  move.l #"HEAD",12(a0)
  move.w bob_höhe,16(a0)          ; Höhe
  move.w bob_bytebreite,d0        ; Breite in Bytes
  lsr.w #1,d0                     ; durch 2 = Breite in Words
  move.w d0,18(a0)                ; und Wordbreite speichern
  lsl.w #4,d0
  move.w d0,20(a0)                ; Breite in Pixeln
  move.w bob_planes,22(a0)        ; Tiefe
  move.w #0,24(a0)                ; 2 Bytes frei
  move.w #0,26(a0)                ; Y-Pos
  move.w #0,28(a0)                ; X-Pos
```

```
  move.w #0,30(a0)              ; No Color
  move.l #"BODY",32(a0)         ; BODY - Name
  move.l bob_planes_size,36(a0) ; Imagelänge
  move.l brush_base,a1          ; Brushspeicher
  move.l puffer+4,d0            ; wieder
  move.l 4,a6                   ; freigeben
  jsr -210(a6)                  ; FreeMem ()
  clr.l brush_base
  lea text6,a0                  ; Ok-Meldung
  bsr printtext
  rts

;--- BOB auf Diskette speichern ---
save_bob:
  bsr savebob
  bra main

savebob:
  move.l bob_planes_base,a1     ; Befindet sich ein BOB im
                                  Speicher
  cmp.l #0,a1                   ; wenn ja, dann weiter
  bne savebob_1
  rts
savebob_1:
;--- BOB-Header auf Disk schreiben ---
  move.l #1006,d2               ; Modus = New
  move.l #filename,d1           ; Filename
  move.l dosbase,a6
  jsr -30(a6)                   ; OPEN ()
  move.l d0,filehd              ; Fileadresse
  tst.l d0                      ; Error ?
  bne ok7                       ; wenn nicht, dann ok7
  lea text8,a0                  ; Errormeldung
  bsr printtext                 ; ausgeben
  rts
ok7:
  move.l filehd,d1              ; Fileadresse
  move.l #bob_header,d2         ; Daten
```

```
  move.l #40,d3                    ; Länge = 40 Bytes
  move.l dosbase,a6
  jsr -48(a6)                      ; Write ()

  move.l filehd,d1                 ; ab hier
  move.l bob_planes_base,d2        ; werden die entpackten
  move.l bob_planes_size,d3        ; Grafikdaten vom BOB
  move.l dosbase,a6                ; gespeichert
  jsr -48(a6)                      ; WRITE ()
  move.l filehd,d1
  move.l dosbase,a6
  jsr -36(a6)                      ; CLOSE ()
;
;--- OK-Meldung ausgeben ---
  lea text9,a0
  bsr printtext
  rts

;---- Brush-Picdaten entpacken ---
start_dekompression:
  move.l body_chunk,a0
  add.l #8,a0                      ; Quelle für Picdaten ist A0
  clr.l d7                         ; D7 = Zähler für eine Spalte-
                                   nanzahl
  clr.l d6                         ; Brushpicdatenzähler
  clr.l d5                         ; D5 = Zähler für Zeilenanzahl
  clr.l d4                         ; D4 = Zähler für Planes

deko_spalte:
  cmp.w bob_bytebreite,d7          ; schon eine Zeile einer Bit-
                                   map entpacked ?
  bmi dekoa                        ; wenn nicht, dann 'dekoa'
  clr.l d7                         ; Spaltenzähler auf Null zu-
                                   rücksetzen
  add.w #1,d4                      ; Bitmapzähler um 1 erniedri-
                                   gen
  cmp.w bob_planes,d4              ; schon eine komplette Zeile
                                   der Grafik
```

```
  bmi deko_spalte                ; entpacked ? - wenn nicht
                                   'deko_spalte'
  clr.l d4                       ; Planeszähler auf Null zu-
                                   rücksetzen
  cmp.l bob_planes_size,d6       ; schon komplette Grafik ent-
                                   packet ?
  bmi deko_spalte                ; wenn nicht, dann weiter ma-
                                   chen
  rts                            ; Ende der Entpackerroutine
;-------- Eine Zeile entpacken ----------------------

dekoa:
  move.l d4,d3                   ; Planenummer nach D3
  mulu #4,d3                     ; mit 4 multiplizieren
  move.l #bitmapadressen,a1      ; Ziel für Picdaten ist A1 (Ta-
                                   belle)
  move.l (a1,d3),a2              ; A2 ist Zeiger auf Bitmapa-
                                   dresse
  bsr bob_unpacker               ; Eine Bitmapzeile entpacken
  move.l a2,(a1,d3)              ; Bitmapposition saven
  bra deko_spalte                ; nächste Zeile

;--------- eigentliche Entpackerroutine ------------
bob_unpacker:
  clr.l d0                       ; Zähler für max. 128 Bytes
  cmp.b #0,packerbyte            ; Brush gepacket ?
  bne gepacked                   ; wenn ja, dann 'gepacked'

;---- eine nicht gepackte Zeile übernehmen ---
  move.w bob_bytebreite,d0       ; Bob-Breite in Bytes nach D0
  sub.w #1,d0                    ; minus 1
  bra deko1loop                  ; und Picdaten übernehmen
;
gepacked:
  move.b (a0)+,d0                ; Befehlsbytes holen
  bmi deko2                      ; wenn negativ, dann 'deko2'
;
```

```
deko1loop:
  move.b (a0)+,(a2)+          ; sonst Picdaten kopieren
  add.w #1,d7                 ; Spaltenzähler plus 1
  add.l #1,d6                 ; Brushdatenzähler plus 1
  dbra d0,deko1loop           ; solange bis Befehlsbyte =
                                Null
  rts
;
deko2:
  neg.b d0                    ; vom Befehlsbyte Vorzeichen
                                wechseln
  move.b (a0)+,d1             ; Füllbyte nach D1
deko2loop:
  move.b d1,(a2)+             ; Füllbyte kopieren
  add.w #1,d7                 ; solange bis Befehlsbyte
  add.l #1,d6                 ; gleich Null ist
  dbra d0,deko2loop
  rts

;--- Gibt einen Text auf dem Bildschirm aus,      ---
;--- der mit einem Nullbyte endet                 ---
;--- Parameter: A0 = Text,                        ---
PrintText:
  bsr printt1
  bsr return
  lea textclear,a0
printt1:
  move.l gfxbase,a6           ; graphics.basis
  move.l #0,d0                ; X-Pos
  move.l #100,d1              ; Y-Pos
  move.l workscreenhd,a1
  add.l #84,a1                ; RastPort
  clr.w d2                    ; Zähler für Anzahl Zeichen
  move.l a0,a2
pt_loop:
  tst.b (a2)+
  beq pt_loop_end
  add.w #1,d2
```

```
  cmp.w #100,d2                    ; Max. 80 Zeichen
  bne pt_loop
pt_loop_end:
  tst.w d2                         ; Kein Text ?
  beq pt_end
  movem.l a0-a1/a6/d2,-(sp)        ; Parameter retten
  jsr -240(a6)                     ; MOVE () - Position setzen
  movem.l (sp)+,a0-a1/a6/d0        ; Parameter holen
  jsr -60(a6)                      ; TEXT () - Text printen
pt_end:
  rts

;---- Auf Returntaste warten ----
return:
  move.b $bfec01,d0
  cmp.b #$77,d0
  bne return
  rts

;--- Parameter für Programm ---
filehd: dc.l 0
puffer: blk.b 12,0
bob_header: blk.b 40,0
brush_base: dc.l 0
chunktabelle: dc.l "BMHD","BODY",0
chunkadresse: dc.l bmhd_chunk,body_chunk
iffchunk: dc.l 0
bmhd_chunk: dc.l 0
body_chunk: dc.l 0
bob_planes_size: dc.l 0
bob_planes_base: dc.l 0
bitmapadressen: blk.l 8,0
bob_planes: dc.w 0
bob_höhe: dc.w 0
bob_bytebreite: dc.w 0
packerbyte: dc.b 0                 ; Dekompressions-Byte
 even
textclear: dc.b "                                ",0
```

```
text2: dc.b "Kann File auf Diskette nicht finden! (ENTER)",0
 even
text3: dc.b "Kein IFF-Standart! (ENTER)",0
 even
text4: dc.b "Datei nicht im ILBM-Format! (ENTER)",0
 even
text5: dc.b "Nicht genug Speicher vorhanden! (ENTER)",0
 even
text6: dc.b "Brush erfolgreich in BOB umgewandelt! (EN-
TER)",0
 even
text8: dc.b "Disk-Error! (ENTER)",0
 even
text9: dc.b "BOB erfolgreich auf Disk gespeichert! (ENTER)",0
 even
;
intbase: dc.l 0
dosbase: dc.l 0
gfxbase: dc.l 0
intname: dc.b "intuition.library",0
 even
dosname: dc.b "dos.library",0
 even
gfxname: dc.b "graphics.library",0
 even
workscreenhd: dc.l 0
workscreen:
 dc.w 0,0,640,256,2,$0701,$8000,15
 dc.l 0,worktitle,0,0
worktitle:
 dc.b "Brushformer V1.0 --- " 10/1990 by J. Schimanski",0
 even
workwindowhd: dc.l 0
workwindow:
 dc.w 0,10,640,246,$0701
 dc.l $120,$2c00
 dc.l gadget,0,0
```

```
workshd: dc.l 0,0
 dc.w 640,256,640,256,15

;--- Gadget für Texteingabe ----
gadget:
 dc.l 0
 dc.w 420,50
 dc.w 160,10
 dc.w 0                          ; flags-ereignisse
 dc.w 2                          ; activasion-flags
 dc.w 4
 dc.l border                     ; Rahmen
 dc.l 0
 dc.l texteingabe
 dc.l 0
 dc.l specialinfo
 dc.w 1
 dc.l 0
texteingabe:
 dc.b 1,0,4
 even
 dc.w 10,-15
 dc.l 0
 dc.l texte
 dc.l 0
texte: dc.b "Filename:",0
 align
border:
 dc.w 0,0
 dc.b 3,3,0,5
 dc.l koordinaten
 dc.l 0
koordinaten:
 dc.w -2,-2,160,-2,160,9,-2,9,-2,-2
specialinfo:
 dc.l filename
 dc.l 0
 dc.w 0,80,0                     ; max. 80 Zeichen
```

```
 dc.w 0,0,0,0,0,0,0
 dc.l 0
;
filename: blk.b 80,0                    ; Textpuffer (80 Bytes)

;--- Menü-Strukturen ---
menü0:
 dc.l 0
 dc.w 0,0,120,10,1
 dc.l menü0text,menü00,0,0
menü0text: dc.b "Transformer",0
 even
;
menü00:
 dc.l menü01
 dc.w 0,0,120,10,$52
 dc.l 0,menü00text,0
 dc.w 0
 dc.l 0,0
menü00text:
 dc.b $7,1,0,0
 dc.w 0,0
 dc.l 0,menü00text0,0

menü00text0:
 dc.b "Load Brush",0
 even
;
menü01:
 dc.l menü02
 dc.w 0,10,120,10,$52
 dc.l 0,menü01text,0
 dc.w 0
 dc.l 0,0
menü01text:
 dc.b $7,1,0,0
 dc.w 0,0
 dc.l 0,menü01text1,0
```

```
menü01text1:
 dc.b "SAVE BOB",0
 even
;
menü02:
 dc.l 0
 dc.w 0,20,120,10,$52
 dc.l 0,menü02text,0
 dc.w 0
 dc.l 0,0
menü02text:
 dc.b $7,1,0,0
 dc.w 0,0
 dc.l 0,menü02text1,0
menü02text1:
 dc.b "EXIT",0
 even

 END
;--- Listingende ---
```

7.9 Einen BOB über den Bildschirm bewegen

So jetzt sind wir soweit auf alles Notwendige eingegangen, daß es an der Zeit ist, endlich mal einen BOB auf dem Bildschirm darzustellen und zu bewegen. Wir werden in diesem Abschnitt auf das ganz einfache Erzeugen von BOBs eingehen. Dafür werden wir einen Screen über die Hardware programmieren und unseren BOB während des Copperinterrupts bewegen. Obwohl das Listing sehr ausführlich dokumentiert ist, folgt zuvor eine kurze Programmbeschreibung:

Da wir für unsere BOB-Darstellung das Betriebssystem ausschalten, warten wir als erstes, in Form einer Warteschleife, bis der Diskmotor ausgelaufen ist. Unbedingt notwendig ist dieses nicht, aber es sieht einfach professioneller aus. Wenn die Warteschleife beendet ist, schalten wir schließlich das System aus.

Um später auch Text auf den Screen ausgeben zu können, öffnen wir jetzt die Graphicsbibliothek und legen uns eine RastPort-Struktur an. Als nächstes wird unser Screen mit den Ausmaßen 320 X 256 in Form einer BitMap-Struktur initialisiert. Für den Screen können bis zu 32 Farben benutzt werden. Was auf keinen Fall vergessen werden darf, ist die BitMap-Struktur in die RastPort-Struktur einzuhängen, ansonsten würde kein Text erscheinen.

Natürlich erzeugen wir unseren Screen mit Hilfe einer Copperliste. Dafür versorgen wir dir Copperregister mit den notwendigen Werten (InitColor und InitCopperMap). Jetzt können wir unsere Copperliste starten und unser Screen erscheint.

Damit wir nicht ein störendes Flimmern vom Mauspointer sehen, schalten wir diesen ab und geben einen Demo-Text auf den Bildschirm aus.

Wie sie ja mitlerweile wissen, benötigt ein Blitterobjekt mehrere Puffer (ShadowMask,CollMask und SaveBuffer). Diese initialiseren wir nun.

Was sie jetzt auch auf keinen Fall vergessen dürfen, ist die Bit-Map-Struktur in die Variable po_bitmap einzutragen, weil diese von der Funktion PrintObjekt () benötigt wird. Ansonsten erscheint später kein Objekt.

Man könnte rein theoretisch jetzt den BOB schon über den Bildschirm bewegen. Doch dieser wäre viel zu schnell. Auch wenn man ihn durch eine Verzögerungsschleife bremsen würde, hätte dies immer noch ein unschönes Flimmern zur Folge. Deswegen bewegen wir unser Objekt während des Copperinterrupts, welchen wir jetzt initialisieren. Damit der Hintergrund nicht vom Objekt während seiner Bewegung zerstört wird, setzen wir in die Objekt-Args-Struktur den Parameter BOBOFF auf den Wert 0.

Auf Druck der rechten Maustaste wird das Programm schließlich beendet.

Für das korrekte Beenden des Programms hängen wir als erstes die alte IRQ-Routine vom System wieder ein, schalten den BOB aus, geben den belegten Speicher wieder zurück (FreeMask, FreeSaveBuffer, ClearBitMap) und schalten die alte Copperliste ein. Zuletzt schließen wir noch die Graphicsbibliothek und schalten das Betriebssystem wieder an.

Hier das dazugehörige Listing:

```
;--- Erzeugt einen Screen über die Hardware      ---
;--- Printet einen Text und bewegt einen BOB     ---
;--- und auf rechte Maustaste erscheint alte     ---
;--- Copperliste und Programm wird beendet       ---
;--- Programmname = MoveBOB                      ---
;
;--- Betriebssystem ausschalten ---
  move.l #600000,d0
motor_aus:
  sub.l #1,d0                    ; warten
  cmp.l #0,d0                    ; bis
```

```
  bne motor_aus                ; Diskmotor aus
  move.l 4,a6                  ; IRQs aus
  jsr -120(a6)                 ; Disable ()
;
  move.l 4,a6                  ; ExecBasis
  lea gfxname,a1               ; Libraryname
  clr.l d0                     ; Version = 0
  jsr -552(a6)                 ; OpenLibrary ()
  move.l d0,gfxbase            ; graphics.basis
;
  lea rastport,a1              ; RastPort-Struktur
  move.l gfxbase,a6            ; graphics.basis
  jsr -198(a6)                 ; InitRastPort ()
;
  lea bitmap,a0                ; BitMap-Struktur
  move.l #5,d0                 ; Depth = 5
  move.l #320,d1               ; 320 breit
  move.l #256,d2               ; 256 hoch
  move.l #1,d3                 ; Speicher reservieren
  bsr initbitmap               ; BitMap init.
;
  lea bitmap,a0                ; BitMap-Struktur
  lea rastport,a1              ; RastPort-Struktur
  move.l a0,4(a1)              ; BitMap in RastPort einhän-
                                 gen
;
  lea colortable,a0            ; Farbtabelle
  lea colorpuffer,a1           ; Copperpuffer
  bsr initcolor                ; Farbregister init.
;
  lea bitmap,a0                ; BitMap-Struktur
  lea copperpuffer,a1          ; Copperpuffer
  clr.l d0                     ; Modus = Normal
  bsr initcoppermap            ; Customregister init.
;
  move.l #copperlist,$dff080   ; Copperlistadresse
  clr.w $dff088                ; Copperliste starten
;
```

```
  move.w #$20,$dff096          ; Sprite-DMA aus
  move.l #spritedatas,$dff120  ; Sprite 0 (Mauszeiger) aus
;
  lea demotext,a0              ; Textadresse
  lea rastport,a1              ; RastPort-Struktur
  move.l gfxbase,a6            ; graphics.basis
  clr.l d0                     ; X-Position
  move.l #100,d1               ; Y-Position
  bsr printtext                ; Text printen
;
;--- Puffer für Hintergrundspeicherung reservieren ---
  lea objektargs,a0            ; ObjektArgs-Struktur
  lea bitmap,a1                ; BitMap-Struktur
  bsr getsavebuffer            ; SaveBuffer reservieren
;
;--- Puffer für Masken reservieren und initialisieren ---
  lea objektargs,a0            ; ObjektArgs-Struktur
  move.l #2,d0                 ; MEMORY + CollMask =
                               ShadowMask
  bsr InitMask                 ; Initialisiere Masken und
                               Puffer
;
;--- BitMap-Struktur in Variable po_bitmap schreiben ---
  move.l #bitmap,po_bitmap
;
;--- CopperIrq einschalten ---
  move.l $6c,oldirqebene       ; alte IRQ-Ebene retten
  move.w $dff01c,intena_buf    ; IRQ-Register retten
  move.l #CopperIrq,$6c        ; eigene IRQ-Routine einhän-
                               gen
  move.w #$7fff,$dff09a        ; alle IRQs aus
  move.w #$c010,$dff09a        ; Copper-IRQ ein
;
;--- Maustaste abfragen ---
main:
  btst #10,$dff016             ; Rechte Maustaste ?
  bne main
  bra exit                     ; Programm beenden
```

```
;--- CopperIRQ-Routine ---
copperirq:
  move.w #$0010,$dff09c          ; CoperIRQ zurücksetzen
  movem.l d0-d7/a0-a6,-(sp)      ; Daten- Adressregister retten
;
  bsr move_bob                   ; BOB bewegen
;
  movem.l (sp)+,d0-d7/a0-a6      ; Register zurückholen
  rte                            ; IRQ-Programm beenden
;

;--- BOB darstellen und bewegen ---
move_bob:
  lea objektargs,a0
  lea speedy,a1
  lea speedx,a2
;
  cmp.w #288,8(a0)               ; XPos
  blt x_kleiner                  ; < 288 ?
  neg.w (a2)                     ; Vorzeichen von X-Pos. ändern

x_kleiner:
  cmp.w #0,8(a0)                 ; XPos
  bge x_größer_gleich
  neg.w (a2)                     ; Vorzeichen von X-Pos ändern

x_größer_gleich:
  move.w (a2),d0                 ; X-Speed
  add.w d0,8(a0)                 ; New-XPos.
;
  cmp.w #246,6(a0)               ; YPos
  blt y_kleiner                  ; < 246 ?
  neg.w (a1)                     ; Vorzeichen von Y-Pos. ändern

y_kleiner:
  cmp.w #0,6(a0)                 ; YPos
  bge y_größer_gleich
```

```
  neg.w (a1)                    ; Vorzeichen von Y-Pos ändern
y_größer_gleich:
  move.w (a1),d0                ; Y-Speed
  add.w d0,6(a0)                ; New-YPos.
;
  bsr printobjekt               ; Objekt darstellen
  rts
;
speedy: dc.w 1                  ; Y-Speed
speedx: dc.w 1                  ; X-Speed

;--- Programm beenden ---
exit:
;
;--- CopperIRQ wieder aus ---
or.w #$8000,intena_buf          ; Set/Clr- Bit setzen
move.w #$7fff,$dff09a           ; alle IRQs aus
move.w intena_buf,$dff09a       ; alte IRQs wieder einschalten
move.l oldirqebene,$6c          ; System-IRQ-Routine einhängen
;
;--- BOB ausschalten ---
lea objektargs,a0               ; ObjektArgs-Struktur
move.b #1,4(a0)                 ; BOB ausschalten
bsr printobjekt                 ; und ausführen
;
;--- BOB-Maskenpuffer wieder freigeben ---
  lea objektargs,a0             ; ObjektArgs-Struktur
  bsr freemask                  ; Maskenpuffer freigeben
;
;--- SaveBuffer-Puffer wieder freigeben ---
  lea objektargs,a0             ; ObjektArgs-Struktur
  lea bitmap,a1                 ; BitMap-Struktur
  bsr freesavebuffer            ; SaveBuffer freigeben
;
  move.l gfxbase,a0             ; graphics.basis
  move.l 38(a0),$dff080         ; alte Copperlistadresse
```

```
  clr.w $dff088                    ; und starten
;
  move.w #$8220,$dff096 ; Sprite-DMA an
;
  lea bitmap,a0                    ; BitMap-Struktur
  bsr clearbitmap                  ; BitMap freigeben
;
  move.l gfxbase,a1                ; graphics.basis
  move.l 4,a6                      ; exec.basis
  jsr -414(a6)                     ; CloseLibrary ()
;


;--- System wieder anschalten ---
  move.l 4,a6                      ; IRQs wieder erlauben
  jsr -126(a6)                     ; Enable ()
  rts                              ; Programmende

;--- SaveBuffer-Speicher wieder freigeben ---
;--- A0 = ObjektArgs; A1 = BitMap ---
FreeSaveBuffer:
  move.w 12(a0),d0                 ; WordWidth
  lsl.w #1,d0                      ; mal 2 = Bytes
  mulu 10(a0),d0                   ; mal Höhe = MapSize
  move.b 5(a1),d1                  ; Screen-Depth nach D1
  and.w #$00ff,d1                  ; Hi-Byte löschen
  mulu d1,d0                       ; Screen-Depth mal MapSize
  move.l 24(a0),a1
  cmp.l #0,a1
  bne fsb_1
  rts
fsb_1:
  clr.l 24(a0)                     ; SaveBuffer-Zeiger löschen
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                     ; FreeMem
  rts
```

```
;---- Speicher für SaveBuffer reservieren ---
;--- A0 = ObjektArgs; A1 = BitMap--
GetSaveBuffer:
  move.w 12(a0),d0              ; WordWidth
  lsl.w #1,d0                   ; mal 2 = Bytes
  mulu 10(a0),d0                ; mal Höhe = MapSize
  move.b 5(a1),d1               ; Screen-Depth nach D1
  and.w #$00ff,d1               ; Hi-Byte löschen
  mulu d1,d0                    ; Screen-Depth mal MapSize
  move.l #$10002,d1             ; Chip + ClearMem
  move.l #$4,a6
  move.l (a6),a6
  movem.l a0,-(sp)
  jsr -198(a6)                  ; AllocMem
  movem.l (sp)+,a0
  tst.l d0
  bne gsb_1
  move.l #-1,d0                 ; Error
  rts
gsb_1:
  move.l d0,24(a0)              ; SaveBuffer-Zeiger init.
  clr.l d0
  rts

;---- ShadowMask-Puffer wieder freigeben --
;--- A0 = ObjektArgs --
FreeMask:
  move.l 20(a0),a1              ; ShadowMask-Zeiger
  cmp.l #0,a1                   ; vorhanden ?
  bne fm_1
  rts

fm_1:
  clr.l 20(a0)                  ; ShadowMask-Zeiger löschen
  move.w 12(a0),d0              ; WordWidth
  lsl.w #1,d0                   ; mal 2 = ByteWidth
  mulu 10(a0),d0                ; mal Höhe = ShadowMask-
Size
```

```
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                    ; FreeMem
  rts

;---- ShadowMask anlegen ----
;--- A0 = ObjektArgs, D0 = Memory/CollMask-Zeiger --
InitMask:
  tst.w d0                        ; Muß ShadowMask-Puffer re-
                                    serviert werden ?
  beq im_1
  movem.l d0,-(sp)
  move.w 12(a0),d0                ; WordWidth
  lsl.w #1,d0                     ; mal 2 = Bytes
  mulu 10(a0),d0                  ; mal Höhe = ShadowMask-
                                    Size
  move.l #$10002,d1               ; Chip und ClearMem
  move.l #$4,a6
  move.l (a6),a6
  movem.l a0,-(sp)
  jsr -198(a6)                    ; AllocMem
  movem.l (sp)+,a0
  tst.l d0
  bne im_1a
  movem.l (sp)+,d0
  move.l #-1,d0                   ; Error
  rts
im_1a:
  move.l d0,20(a0)                ; ShadowMask-Adresse spei-
                                    chern
  movem.l (sp)+,d0
  cmp.w #2,d0
  bne im_1
  move.l 20(a0),28(a0)
im_1:
  move.l #$dff000,a6
  move.l #$0dfc0000,$40(a6)       ; BLTCON0: A + B = D
  move.l #-1,$44(a6)              ; First/Last Mask
```

```
  clr.l $62(a6)          ; Modulowert von Quelle B
  clr.w $66(a6)          ; Modulowert von Ziel  D
  move.w 12(a0),d5       ; Breite in Words
  move.w 10(a0),d6       ; Höhe in Pixel
  move.w d5,d0           ; Breite nach D0
  lsl.w #1,d0            ; mal 2 = Breite in Bytes
  mulu d6,d0             ; D0 = ImageMapSize
  and.w #$3ff,d6         ; Blittersize errechnen
  lsl.w #6,d6            ; D5 = Breite in Words
  and.w #$3f,d5          ; D6 = Höhe in Pixel
  add.w d6,d5            ; D5 ist jetzt Blittersize
  move.w 14(a0),d1       ; Anzahl Planes nach D1
  sub.w #1,d1            ; minus 1, wegen DBra
  move.l 20(a0),a1       ; A1 = ShadowMask (Ziel D
                           und Quelle B)
  move.l 16(a0),a2       ; A2 = Image-Zeiger
initmask_loop:
  move.l a2,$50(a6)      ; Anfangsadresse von Quelle
                           A
  move.l a1,$54(a6)      ; Anfangsadresse von Ziel  D
  move.l a1,$4c(a6)      ; Anfangsadresse von Quelle
                           B = Ziel D
  move.w d5,$58(a6)      ; BLTSIZE und Blitteroperati-
                           on starten
initmask_wait:
  btst #14,$2(a6)        ; Blitter fertig?
  bne initmask_wait
  add.l d0,a2            ; nächste ImageMap
  dbra d1,initmask_loop
  clr.l d0               ; No Errors
  rts

;--- ein Objekt auf Bildschirm printen
;--- A0 = Objektargs ; po_bitmap = Zeiger auf Bitmap-Struktur
;--- Objekte nicht höher und/oder breiter als Bitmap
;--- Diese Routine kann PC-Relative assembliert werden
PrintObjekt:
  move.l po_bitmap(pc),a1
```

```
  move.l #$dff000,a2          ; Chip-Basisadresse $DFF000
  move.w (a1),d6
  lsr.w #1,d6                 ; D6 = Screenbreite in Words
  move.w 2(a1),d7             ; D7 = Screenhöhe
;
  move.w 6(a0),d0             ; Ypos nach D0
  cmp.w 2(a1),d0              ; If YPos >= Screenhöhe then
                                ende

  bge po_clipping
  move.w 10(a0),d1            ; Höhe des Objekts nach D1
  neg.w d1                    ; negativ machen
  cmp.w d1,d0                 ; If YPos <= D1 then ende
  ble po_clipping
  move.w 8(a0),d0             ; X-Pos. nach D0
  move.w (a1),d1              ; Zeilenbreite in Bytes
  lsl.w #3,d1                 ; mal 8 = Zeilenbreite in Pixel
  cmp.w d1,d0                 ; If XPos >= Zeilenbreite then
                                ende

  bge po_clipping
  move.w 12(a0),d1            ; Objektbreite nach D1
  sub.w #1,d1                 ; ein Word abziehen
  lsl.w #4,d1                 ; mal 16 = Breite in Pixel
  neg.w d1                    ; Objektbreite jetzt negativ
  cmp.w d1,d0                 ; If XPos <= D1 then ende
  ble po_clipping
;
  cmp.b #1,4(a0)              ; BOBOff ?
  bne po_bobon
po_bob_aus:
  bsr po_writehintergrund
  clr.b 5(a0)                 ; BOB nicht mehr init.
  rts
po_bobon:
  tst.b 4(a0)                 ; Objekt normal anschalten ?
  bne po_bobon_xa
  bsr po_writehintergrund
  bsr po_readhintergrund      ; Hintergrund speichern
```

```
  bsr po_writeobjekt              ; Objekt in Hintergrund printen
  rts                             ; Ende
po_bobon_xa:
  cmp.b #2,4(a0)                  ; Write BOB only
  bne po_bobon_xb
  bsr po_writeobjekt              ; Hintergrund zurückschreiben
  rts

po_bobon_xb:
  cmp.b #3,4(a0)                  ; Write Hintergrund only
  bne po_bobon_xc
  bsr po_writehintergrund
  rts
po_bobon_xc:
  cmp.b #4,4(a0)                  ; Read Hintergrund only
  bne po_bobon_end
  bsr po_readhintergrund
po_bobon_end:
  rts
po_clipping:
  tst.b 4(a0)                     ; Objekt normal anschalten ?
  beq po_bob_aus
  cmp.b #1,4(a0)
  beq po_bob_aus
  cmp.b #3,4(a0)
  beq po_bob_aus
  rts
;--- Masken/Offset/Blittersize berechnen
;--- D0 = Y, D1 = X Position
;--- Rückgabe: D0 = Blitterhöhe, D1 = Blitterbreite
;          D2 = PositionOffset, D5 = Shiftwert
;          D4 = muß zum Bitmapoffset addiert werden
po_parameter:
  tst.w d0                        ; Y positiv ?
  bpl po_para_1                   ; wenn ja, dann verzweigen
  move.w 12(a0),d2                ; Breite Objekt Words
```

```
  lsl.w #1,d2                ; in Bytes
  move.w d0,d3
  neg.w d3
  mulu d3,d2
  add.w 10(a0),d0            ; plus höhe Objekt
  bra po_para_x
po_para_1:
  move.w d7,d2               ; Screenhöhe
  sub.w d0,d2                ; minus y-pos
  move.w d2,d0               ; D0 = ergebnis
  clr.l d2
  cmp.w 10(a0),d0            ; minus höhe
  bmi po_para_x              ; negativ ?
  move.w 10(a0),d0           ; wenn positiv dann normale
                               Height
po_para_x:                   ; D0 = Blitterhöhe , D2 = Y-
                               Offset
  tst.w d1                   ; X positiv ?
  bpl po_para_2              ; wenn ja, dann verzweigen
  neg.w d1                   ; X-pos jetzt positiv
  move.w d1,d4               ; x-pos nach D4
  lsr.w #4,d1                ; durch 16 teilen
  move.w 12(a0),d5           ; Objektbreite nach D5
  sub.w d1,d5
  clr.l d3
  move.w d1,d3
  lsl.w #1,d3                ; D3 = X-Offset
  add.l d3,d2                ; D2 = Position-Offset
  move.w d5,d1               ; D1 = Blitterbreite
  and.w #15,d4
  clr.w d5
  tst.w d4
  beq po_para_44
  move.w #16,d5
  sub.w d4,d5                ; D5 = real Shiftwert
  subq.w #1,d4
  move.w #$ffff,d3
```

```
po_para_shift:
  lsr.w #1,d3
  dbra d4,po_para_shift
  move.w d3,$44(a2)          ; FirstMask
  move.w d3,$46(a2)
  cmp.w #1,d1
  beq po_para_7
  move.w #$ffff,$46(a2)      ; LastMask
po_para_7:
  move.l #2,d4               ; Bitmapoffset -2 Bytes
  rts
po_para_2:                   ; X-Pos ist positiv
  move.w d1,d5               ; X-pos nach D5
  and.w #15,d5               ; D5 = Shiftwert
  lsr.w #4,d1                ; X-Pos durch 16
  move.w d6,d4               ; Screenbreite nach D4
  sub.w d1,d4
  move.w d4,d1               ; D1 = Blitterbreite
  cmp.w 12(a0),d1            ; Ergebnis - Objektbreite
  bmi po_para_3              ; hiernach normal weiter
  move.w 12(a0),d1           ; Blitterbreite
po_para_44:
  move.l #-1,$44(a2)         ; First/LastMask
  clr.l d4                   ; Bitmapoffset + 0 Bytes
  rts
po_para_3:
  tst.w d5
  beq po_para_44
  move.w d5,d3
  subq.w #1,d3
  move.w #$ffff,d4
po_para_shifta:
  lsl.w #1,d4
  dbra d3,po_para_shifta
  move.w d4,$44(a2)          ; FirstMask
  move.w d4,$46(a2)
  cmp.w #1,d1                ; Blitterbreite = 1
  beq po_para_4
```

```
  move.w #$ffff,$44(a2)          ; LastMask
po_para_4:
  clr.l d4                       ; Bitmapoffset + 0 Bytes
  rts

;------ Hintergrund wieder printen ----
po_writehintergrund:
  tst.b 5(a0)                    ; wurde ein Hintergrund
                                   schon gelesen ?
  bne po_writehg_x
  rts
po_writehg_x:
  move.w (a0),d0                 ; OldY
  move.w 2(a0),d1                ; OldX
  bsr po_parameter
  move.l #-1,$44(a2)             ; First/LastMask
  move.l #$09f00000,$40(a2)      ; BLTCON0/1
  move.l 24(a0),a3               ; A3 = real Quelle A
  add.l d2,a3
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)              ; ZModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)              ; AModulo
  clr.l d3
  tst.w 2(a0)
  bmi po_writehg_nox
  move.w 2(a0),d3                ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                    ; X-Offset
po_writehg_nox:
  tst.w (a0)
  bmi po_writehg_noy
  move.w (a0),d4
```

```
  mulu d6,d4
  lsl.l #1,d4                    ; Y-Offset
  add.l d4,d3                    ; D3 = Bitmapoffset
po_writehg_noy:
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                    ; D4 = Map-Size vom Objekt
  clr.w d5
  move.b 5(a1),d5
  subq.w #1,d5                   ; D5 = Loop-zähler
  and.w #$3ff,d0                 ; Blittersize errechnen
  lsl.w #6,d0                    ; D1 = breite in Words
  and.w #$3f,d1                  ; D0 = höhe in Pixel
  add.w d0,d1                    ; D1 ist jetzt Blittersize
po_writehg_loop:
  move.l a3,$50(a2)              ; Quelle A
  move.l (a4)+,a5
  add.l d3,a5
  move.l a5,$54(a2)              ; Ziel D
  move.w d1,$58(a2)              ; Blitter starten
po_writehg_wait:
  btst #14,$2(a2)                ; Bit BBusy testen
  bne po_writehg_wait            ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_writehg_loop
rts

;------ Hintergrund speichern --------
po_readhintergrund:
  move.b #1,5(a0)                ; Init = 1, Hintergrund schon
                                   mal gelesen
  move.w 6(a0),0(a0)             ; Ypos nach OldYpos
  move.w 8(a0),2(a0)             ; Xpos nach OldXpos
  move.w (a0),d0                 ; Y
  move.w 2(a0),d1                ; X
  bsr po_parameter
  move.l #-1,$44(a2)             ; First/LastMask
  move.l #$09f00000,$40(a2)      ; BLTCON0/1
```

```
  move.l 24(a0),a3          ; A3 = Ziel D
  add.l d2,a3
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)         ; AModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)         ; ZModulo
  clr.l d3
  tst.w 2(a0)
  bmi po_readhg_nox
  move.w 2(a0),d3           ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3               ; X-Offset
po_readhg_nox:
  tst.w (a0)
  bmi po_readhg_noy
  move.w (a0),d4
  mulu d6,d4
  lsl.l #1,d4               ; Y-Offset
  add.l d4,d3               ; D3 = Bitmapoffset
po_readhg_noy:
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4               ; D4 = Map-Size vom Objekt
  clr.w d5
  move.b 5(a1),d5
  subq.w #1,d5              ; D5 = Loop-zähler
  and.w #$3ff,d0            ; Blittersize errechnen
  lsl.w #6,d0               ; D1 = breite in Words
  and.w #$3f,d1             ; D0 = höhe in Pixel
  add.w d0,d1               ; D1 ist jetzt Blittersize
po_readhg_loop:
  move.l a3,$54(a2)         ; Ziel D
  move.l (a4)+,a5
```

```
  add.l d3,a5
  move.l a5,$50(a2)              ; Quelle A
  move.w d1,$58(a2)              ; Blitter starten
po_readhg_wait:
  btst #14,$2(a2)                ; Bit BBusy testen
  bne po_readhg_wait             ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_readhg_loop
  rts

;--- Objekt Daten in Bitmap kopieren ---
po_writeobjekt:
  move.w 6(a0),d0                ; Y
  move.w 8(a0),d1                ; X
  bsr po_parameter
  lsl.w #8,d5
  lsl.w #4,d5                    ; korrekter Shiftwert
  move.w d5,$42(a2)              ; BLTCON1
  add.w #$0fca,d5
  movem.l d5,-(sp)
  move.w d5,$40(a2)              ; BLTCON0
  move.l 20(a0),a5
  add.l d2,a5                    ; A5 = Quelle A (real Shadow-
                                 Mask)
  move.l 16(a0),a3
  add.l d2,a3                    ; A3 = Quelle B (real Image)
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)              ; ZModulo
  move.w d5,$60(a2)              ; CModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)              ; AModulo
  move.w d5,$62(a2)              ; BModulo
  clr.l d3
```

```
  tst.w 8(a0)
  bmi po_writeo_nox
  move.w 8(a0),d3                ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                    ; X-Offset
po_writeo_nox:
  tst.w 6(a0)
  bmi po_writeo_noy
  move.w 6(a0),d5
  mulu d6,d5
  lsl.l #1,d5                    ; Y-Offset
  add.l d5,d3
po_writeo_noy:
  sub.l d4,d3                    ; D3 = Bitmapoffset
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                    ; D4 = Map-Size vom Objekt
  move.w 14(a0),d5
  subq.w #1,d5                   ; D5 = Loop-zähler
  and.w #$3ff,d0
  and.w #$3f,d1
  lsl.w #6,d0                    ; D1 = breite in Words
  add.w d0,d1                    ; D1 ist jetzt Blittersize
po_writeo_loop:
  move.l a5,$50(a2)              ; Quelle A (ShadowMask)
  move.l a3,$4c(a2)              ; Quelle B (Image)
  move.l (a4)+,a6
  add.l d3,a6
  move.l a6,$48(a2)              ; Quelle C (BitMap)
  move.l a6,$54(a2)              ; Ziel  D (BitMap)
  move.w d1,$58(a2)              ; Blitter starten
po_writeo_wait:
  btst #14,$2(a2)                ; Bit BBusy testen
  bne po_writeo_wait             ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_writeo_loop
  movem.l (sp)+,d5               ; BLTCON0 wert holen
  move.w 14(a0),d0               ; Depth BOB nach D0
```

```
  cmp.b 5(a1),d0          ; = Anzahl Planes Screen ?
  beq po_writeo_end       ; Wenn ja, dann Ende
  sub.b 5(a1),d0          ; Planes-Anzahl abziehen
  neg.b d0                ; Positiv machen
  subq.w #1,d0
  sub.w #$0400,d5         ; DMA-Kanal B = aus
  move.w d5,$40(a2)       ; BLTCON0
  clr.w $42(a2)           ; No Shift B
  clr.w $72(a2)           ; Clear Datenregister B (Figur).

po_writeo_loop2:
  move.l a5,$50(a2)       ; Quelle A (ShadowMask)
  move.l (a4)+,a6
  add.l d3,a6
  move.l a6,$48(a2)       ; Quelle C (BitMap)
  move.l a6,$54(a2)       ; Ziel D (BitMap)
  move.w d1,$58(a2)       ; Blitter starten
po_writeo_wait2:
  btst #14,$2(a2)         ; Bit BBusy testen
  bne po_writeo_wait2     ; wenn Null, dann Blitterende
  dbra d0,po_writeo_loop2
po_writeo_end:
  rts
po_bitmap: dc.l 0


;--- Gibt einen Text auf dem Bildschirm aus,      ---
;--- der mit einem Nullbyte endet                 ---
;--- Parameter: A0 = Text, A1 = RastPort,         ---
;---            A6 = Graphics-Basis               ---
;---            D0 = X-Pos., D1 = Y-Position      ---
PrintText:
  clr.w d2                ; Zähler für Anzahl Zeichen
  move.l a0,a2
pt_loop:
  tst.b (a2)+
  beq pt_loop_end
  add.w #1,d2
```

```
  cmp.w #100,d2                ; Max. 80 Zeichen
  bne pt_loop
pt_loop_end:
  tst.w d2                     ; Kein Text ?
  beq pt_end
  movem.l a0-a1/a6/d2,-(sp)    ; Parameter retten
  jsr -240(a6)                 ; MOVE () - Position setzen
  movem.l (sp)+,a0-a1/a6/d0    ; Parameter holen
  jsr -60(a6)                  ; TEXT () - Text printen
pt_end:
  rts

;--- BitMap-Pointer in CopperList übertragen ---
;--- A0 = BitMap ; A1 = CopperPuffer ; D0 = Modus ---
InitCopperMap:
  move.w (a0),d1               ; Bytes pro Zeile nach D1
  sub.w #40,d1                 ; minus 40 Bytes (320 Pixel)
  cmp.w #$8000,d0              ; Modus = Hires ?
  bne icm_1
  sub.w #40,d1                 ; nochmal minus 40 Bytes
                               (insgesammt -640 Pixel)
icm_1:
  cmp.w #$04,d0                ; Modus = Interlace ?
  bne icm_2
  sub.w #40,d1                 ; auch minus 40 Bytes (ins-
                               gesammt -640 Pixel)
icm_2:
  move.w #$0108,(a1)+          ; BPL1MOD
  move.w d1,(a1)+              ; Modulo-Wert ungerade Pla-
                               nes
  move.w #$010a,(a1)+          ; BPL2MOD
  move.w d1,(a1)+              ; Modulo-Wert gerade Planes
  move.b 5(a0),d1              ; Depth nach D1
  lsl.w #8,d1                  ; Korekten
  lsl.w #5,d1                  ; Depth-Wert
  lsr.w #1,d1                  ; ermitteln
  add.w #$0200,d1              ; Color setzen
  add.w d0,d1                  ; Modus setzen
```

```
  move.w #$0100,(a1)+          ; BPLCON0
  move.w d1,(a1)+              ; setzen
  moveq #5,d0                  ; max. Tiefe minus 1
  move.w #$00e0,d1             ; Hi-Word vom ersten Map-
                                 Pointer
  add.l #8,a0                  ; Erster Pointer-Zeiger
icm_loop:
  move.w d1,(a1)+              ; HI-Word vom Pointer
  move.w (a0)+,(a1)+           ; setzen
  add.w #2,d1                  ; Lo-Word ermitteln
  move.w d1,(a1)+              ; und
  move.w (a0)+,(a1)+           ; setzen
  add.w #2,d1                  ; Hi-Word ermitteln
  dbra d0,icm_loop
  rts

;---- ColorMap in CopperListe übertragen ---
;--- A0 = ColorTable ; A1 = CopperPuffer ---
InitColor:
  move.w #$180,d1              ; erstes Farbregister
  move.w #31,d0                ; Anzahl Farben
ic_loop:
  move.w d1,(a1)+              ; Farbregister in CopperList
  move.w (a0)+,(a1)+           ; dann der Farbwert in Cop-
                                 perList
  add.w #2,d1                  ; nächstes Farbregister
  dbra d0,ic_loop
  rts

;--- BitMap-Struktur initialisieren und
;--- Speicher für Maps reserverieren
;--- D0 = Depth, D1 = Width, D2 = Height, D3 = Memory,
;--- A0 = BitMap
InitBitMap:
  move.w d1,d4                 ; Breite nach D4
  and.w #15,d4                 ; Rest ermitteln
  tst.w d4                     ; Rest vorhanden ?
```

```
  beq ibm_1                 ; Wenn nicht, dann weiter -
                              sonst Error
  move.l #-1,d0             ; Error: Breite nicht durch 16
                              teilbar
  rts
ibm_1:
  lsr.w #4,d1               ; Breite durch 16 teilen
  lsl.w #1,d1               ; mal 2 = Anzahl Bytes einer
                              Zeile
  move.w d1,(a0)            ; und in BitMap-Struktur
                              speichern
  move.w d2,2(a0)           ; Höhe in BitMap-Struktur
                              speichern
  move.w d0,4(a0)           ; Anzahl Planes speichern
  tst.w d3                  ; Muß Speicher für BitMaps
                              reserviert werden ?
  bne ibm_2                 ; Wenn nicht, dann Ende,
                              sonst weiter
  clr.l d0                  ; No Errors
  rts                       ; Ende
ibm_2:
  move.w d0,d4              ; Depth nach D4
  sub.w #1,d4               ; minus 1
  lea 8(a0),a1              ; BitMap nach a1
ibm_clear_loop:             ; Pointer-Zeiger
  clr.l (a1)+               ; löschen
  dbra d4,ibm_clear_loop

  mulu d2,d1                ; BytePerRow x Höhe = Size
                              von einer Map
  move.w d0,d2              ; D2 = Anzahl Planes
  move.l d1,d0              ; D0 = ByteSize
  move.l #$10002,d1         ; D1 = CHIP + FreeMem
  sub.w #1,d2               ; Anzahl Planes minus 1, we-
                              gen DBRA
  add.l #8,a0               ; Anfang BitMap-Zeiger
  move.l a0,a2              ; auch nach A2
```

```
ibm_loop:
  move.l #$4,a6
  move.l (a6),a6
  movem.l d0-d2/a0-a2,-(sp)
  jsr -198(a6)                    ; AllocMem
  tst.l d0                        ; konnte Speicher reserviert
                                    werden ?
  bne ibm_l1                      ; Wenn ja, dann weiter
  movem.l (sp)+,d0-d2/a0-a2
ibm_free_loop:
  move.l (a2),a1                  ; Zeiger auf BitMap holen
  cmp.l #0,a1                     ; Null ?
  bne ibm_l2                      ; wenn ja, dann ende
  move.l #-2,d0                   ; Error
  rts                             ; Ende
ibm_l2:
  clr.l (a2)+
  movem.l d0/a2,-(sp)
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                    ; FreeMem
  movem.l (sp)+,d0/a2
  bra ibm_free_loop
ibm_l1:
  move.l d0,d5                    ; Zeiger auf BitMap nach D5
  movem.l (sp)+,d0-d2/a0-a2
  move.l d5,(a0)+
  dbra d2,ibm_loop
  clr.l d0
  rts

;--- Speicher der Maps wieder freigeben in
;--- einer BitMap-Struktur
;--- A0 = BitMap
ClearBitMap:
  clr.w d1
  move.b 5(a0),d1                 ; Depth nach D1
  sub.w #1,d1
```

```
  move.w (a0),d0                ; BytePerRow
  mulu 2(a0),d0                 ; mal Höhe = MapSize
  add.l #8,a0
  move.l #$4,a6
  move.l (a6),a6
cbm_loop:
  move.l (a0),a1                ; Map-Pointer
  cmp.l #0,a1
  beq cbm_l1
  clr.l (a0)+
  movem.l d0-d1/a0/a6,-(sp)
  jsr -210(a6)                  ; FreeMem
  movem.l (sp)+,d0-d1/a0/a6
cbm_l1:
  dbra d1,cbm_loop
  rts
;--- Parameter ---
;
;--- Objektdaten ---
ObjektArgs:
dc.w 0,0                        ; OLDY,OLDX
dc.b 0,0                        ; BOBOFF = Normal , Init = 0
dc.w 100,20                     ; YPos,XPos
dc.w 10,3,2                     ; Höhe = 10, Wordbreite = 3,
                                  Tiefe = 2
dc.l image                      ; Adresse der BOB-Daten
dc.l 0,0,0                      ; ShadowMask/SaveBuf-
                                  fer/CollMask = Null
;
image:                          ; Imagedaten vom BOB
;--- BitMap 1 ---
dc.w %0000000000000011,%1110000000000000,$0     ;letztes
                                                 Word
dc.w %0000000000000111,%1100000000000000,$0     ; muß
dc.w %0000000000001111,%1000000000000000,$0     ; immer
dc.w %0000000000011111,%0000000000011000,$0     ; Null
dc.w %0000000000111110,%0000000000011000,$0     ; sein
dc.w %1111000001111100,%0000000000000000,$0
```

```
dc.w %0111100011111000,%0110010001000000,$0
dc.w %0011110111110000,%1001011011000000,$0
dc.w %0001111111100000,%1111010101000000,$0
dc.w %0000111111000000,%1001010001000000,$0
;--- BitMap 2 ---
dc.w %0000000000000011,%1011100000000000,$0     ;letztes
                                                 Word
dc.w %0000000000000111,%0111000000000000,$0     ; muß
dc.w %0000000000001110,%1110000000000000,$0  ;immer
dc.w %0000000000011101,%1100000000000000,$0     ; Null
dc.w %0000000000111011,%1000000000000000,$0     ; sein
dc.w %1100110001110111,%0000000000000000,$0
dc.w %0110011011101110,%0110000000011000,$0
dc.w %0011001111011100,%1001000000011000,$0
dc.w %0001101110111000,%1111000000011000,$0
dc.w %0000111111110000,%1001000000011000,$0
;
oldirqebene: dc.l 0
intena_buf: dc.w 0
;
gfxbase: dc.l 0                    ; graphics.basis
gfxname: dc.b "graphics.library",0 ; Libraryname
 even
bitmap: blk.b 40,0                 ; BitMap-Struktur
rastport: blk.b 100,0              ; RastPort-Struktur
colortable:
 dc.w 0,1000,200,600,500,1040,2340,1234,4000,12,456,876
 blk.w 20,100                      ; 32 Farben
spritedatas: blk.l 3,0
copperlist:                        ; ab hier Copperliste
 dc.w $8e,$3081                    ; DIWSTRT
 dc.w $90,$30c1                    ; DIWSTOP
 dc.w $92,$38                      ; DDFSTRT
 dc.w $94,$d0                      ; DDFSTOP
colorpuffer:
 blk.b 128,0                       ; Farbregister
```

```
copperpuffer:
 blk.b 60,0                    ; Screen-Customregister
 dc.w $ff01,$fff4              ; Wait 255
 dc.w $ffdf,$fffe              ; Wait maximale Pos.
 dc.w $0101,$fffe              ; Wait 1 (= 256)
 dc.w $3001,$fffe              ; max. $3801
 dc.w $009c,$8010              ; CopperIRQ auslösen
 dc.l $fffffffe                ; Copperlistende
demotext:
 dc.b "Dieses ist ein Blitter-Objekt (BOB)",0
 even

 END
;--- Listingende ---
```

7.10 Blitzschnelle, flackerfreie BOBs (DoubleBuffering)

Bei der Darstellung von Blitterobjekten kann es passieren, daß sich die Zugriffszeiten des Blitters mit denen des Rasterstrahls überschneiden. Besonders bei großen Objekten, und je näher sie sich zum oberen Bildschirmrand befinden. Immer wenn dieser Fall eintritt, kommt es zu einem Flackern und Ruckeln des Objekts. Um dies zu vermeiden, werden die BOBs doppelt gepuffert (double buffering). Beim Double-Buffering werden mindestens zwei BitMap-Strukturen von gleicher Größe und Aussehen benötigt. Dabei wird eine BitMap-Struktur über die Copperliste aktiviert, die andere liegt nicht sichtbar für den Benutzer im Speicher. Soll jetzt ein Objekt dargestellt werden, so wird dieses im unsichtbaren Hintergrund aufgebaut. Während des Copper- oder Rasterinterrupts wird die unsichtbare BitMap über die Copperliste aktiviert. Das führt zu einem blitzschnellen und ruckfreien Erscheinen des Objekts.

Da wir das Umschalten der BitMaps während des Copper- bzw. Rasterstrahlinterrupts vornehmen, um eine absolut ruckfreie Animation zu gewährleisten, kann sich ein Objekt maximal 50 Punkte in der Sekunde bewegen, da die Interrupts ebenso oft aufgerufen werden. Wollen sie die Animation beschleunigen, so müssen sie das Objekt halt um zwei oder mehr Punkte bewegen.

Das Double-Buffering kann auf mehrere Arten programmiert werden:

1. Methode:

Es werden zwei BitMap-Strukturen angelegt (Hintergründe), die gleichgroß sind und gleiches Aussehen haben, also absolut identisch sind. Wir nennen die beiden BitMap-Strukturen mal BitMap A und BitMap B. Als erstes ist BitMap A aktiviert. Nun schreiben wir von allen Objekten den Hintergrund in die unsichtbare BitMap (BitMap B) an den alten Positionen zurück, retten den Hintergrund an den neuen Positionen und schreiben schließlich das eigentliche Objekt an den neuen Positionen in den unsichtbaren Hintergrund. Jetzt vertauschen wir die BitMaps, so daß BitMap B aktiviert wird und BitMap A unsichtbar ist. Das Vertauschen erledigen wir während des Copperinterrupts, damit die Bewegung auch absolut ruckfrei wird. Nun müssen wir noch die gesamte aktive BitMap (B) in die unsichtbare (A) kopieren, damit die beiden Hintergründe wieder absolut identisch sind. Wollen wir das Objekt jetzt ein weiteres Mal bewegen, so müssen alle Schritte erneut wiederholt werden.

Damit sie das ganze auch besser verstehen, hier alles nochmal in Kurzform:

- zwei identische BitMap-Strukturen anlegen
- erste BitMap-Struktur aktivieren

Loop:

- Hintergrund an alten Positionen zurückschreiben
- Hintergrund an neuen Positionen retten
- Objekte an neuen Positionen in Hintergrund schreiben
- BitMap-Strukturen während des Copperinterrupts vertauschen

- aktive BitMap in unsichtbare BitMap kopieren
- Objekte bewegen (Positionen ändern)
- wieder zum Anfang (Loop)

Vorteil:

Absolut flacker- und ruckfreie Bewegung der Objekte. Relativ einfach zu programmieren.

Nachteil:

Nicht sehr schnell. Nur für kleine Anzahl von kleinen Objekten geeignet, ansonsten großer Speicherbedarf. Großer Aufwand für die Hintergrundaufbearbeitung nötig.

2. Methode:

Es werden wieder zwei gleiche BitMap-Strukturen mit gleichem Aussehen angelegt (BitMap A und B). Um jetzt ein Objekt erscheinen zu lassen, bzw. zu bewegen, retten wir den Hintergrund aus der unsichtbaren BitMap (B). Jetzt schreiben wir das eigentliche Objekt in den ebenfalls unsichtbaren Hintergrund (BitMap B). Wir vertauschen wieder die BitMap-Strukturen während des Copperinterrupts, so daß BitMap B aktiviert ist und BitMap A unsichtbar. Als letztes Schreiben wir den alten Hintergrund in die unsichtbare BitMap (A) wieder zurück. Das war eigentlich schon alles. Wie sie sehen, ist der Aufwand fast derselbe, nur daß wir das Kopieren der einen BitMap-Struktur in die andere weglassen. Damit ist diese Methode gegenüber der ersten um einiges schneller.

Kurzformbeschreibung:

- zwei identische BitMap-Strukturen anlegen
- erste BitMap-Struktur aktivieren

Loop:

- Hintergrund an neuen Positionen aus unsichtbarer BitMap retten
- Objekte in unsichtbare BitMap-Struktur kopieren

- BitMap-Strukturen während des Copperinterrupts vertauschen
- alten Hintergrund in unsichtbare BitMap zurückschreiben
- Objekte bewegen (Positionen ändern)
- wieder zum Anfang (Loop)

Vorteil:

Die selben wie bei der ersten Methode, jedoch um einiges schneller.

Nachteil:

Großer Aufwand für die Hintergrundaufbearbeitung. Bei Animation eines Objekts mit verschieden großen Ausmaßen einer Sequence ist diese Methode nicht mehr anwendbar. Bei Animation eines Objkets müssen alle Sequencen gleichgroß sein.

3. Methode:

Die Methode, auf die wir gleich eingehen, werden wir als Beispiel im nächsten Abschnitt verwenden. Denn es ist von allen wohl die einfachste und schnellste Methode.

Es werden diesmal drei, statt zwei BitMap-Strukturen angelegt, die alle absolut identisch sind. Die dritte wird dabei für die Hintergrundaufbearbeitung benutzt. Sie werden sicherlich denken, was das für ein Speicher kostet! Aber weit gefehlt. Denn damit sparen sie das lästige Anlegen der SaveBuffer-Speicher mit der Routine GetSaveBuffer (). Denn wenn sie zum Beispiel 10 Objekte darstellen wollen, kostet das in der Regel mehr Speicher, als wenn sie einen dritten Hintergrund anlegen. Außerdem sparen sie auch enorme Bitterzeit, da sie das Objekt nur in den Hintergrund kopieren brauchen. Das zusätzliche Retten und wieder Zurückschreiben des Hintergrundes entfällt.

Wollen sie ein Objekt bewegen, so schreiben sie die Grafikdaten des Objekts in die unsichtbare BitMap (B) und aktivieren diese während des Copperinterrupts, so daß BitMap A jetzt unsichtbar ist. Damit der unsichtbare Hintergrund auch wieder seinen Ausgangszustand besitzt, kopieren sie die BitMap C in die unsichtbare BitMap (A). Das war schon alles.

Kurzformbeschreibung:

- drei identische BitMap-Strukturen anlegen
- erste BitMap über Copperliste aktivieren

Loop:

- Objekte in unsichtbare BitMap-Struktur kopieren
- BitMap-Strukturen eins und zwei im Copper-IRQ vertauschen
- BitMap-Struktur drei in unsichtbare kopieren (CopyBitMap)
- Objekte bewegen (Position, Aussehen etc.)
- wieder zum Anfang (Loop)

Nachteile:

Anlegen einer dritten BitMap-Struktur notwendig. Eventuell zu hoher Speicherbedarf (nur bei kleiner Anzahl von Objekten).

Vorteile:

Sehr schnelle Ausführungszeiten (Animation) und einfach zu programmieren. Sehr gut für Animationen von Objekten mit verschieden großen Sequencen geeignet.

Das waren jetzt nur drei Methoden. Es gibt noch einige mehr, die aber vom Prinzip her auf diese aufbauen. Unser Beispielprogramm, das wir im nächsten Abschnitt programmieren, werden wir nach der dritten Methode schreiben.

7.11 Collisionen zwischen BOBs

Ein Blitterobjekt darzustellen bzw. zu animieren ist ja ganz schön. Aber was nützt einem dies bei der Spieleprogrammierung, wenn eine Figur durch eine Mauer läuft, wo sie eigentlich hätte anhalten sollen?

In diesem Fall wäre es erforderlich gewesen, eine Berührung (Collision) mit der Mauer abzufragen und entsprechend darauf zu reagieren. Um eine Collision zwischen zwei Objekten festzustellen, müssen die Positionen miteinander verglichen werden. Überschneiden sich diese, so muß in diesem Bereich noch jeder Punkt des einen Objekts mit dem anderen abgefragt werden. Liegen diese übereinander, so fand eine Collision statt.

Damit die Collisionsabfrage auch schnell vonstatten geht, wird eine ODER-Maske von allen BitMaps eines Objekts angelegt. Die schon bereits erwähnte CollMask. Sie besitzt dasselbe Aussehen wie die ShadowMask und wird deshalb in der Regel auch auf die gleiche Adresse gesetzt. Wir brauchen jetzt nur noch eine Routine, die die Positionen und ggf. die Collisionsmasken miteinander vergleicht und uns meldet, ob eine Collision vorliegt oder nicht.

Selbstverständlich folgt auch an dieser Stelle ein Listing einer entsprechenden Routine, welche sich Collision () nennt. Sie benötigt in den Adressregistern A0 und A1 die Adressen der beiden ObjektArgs- Strukturen von den entsprechenden Objekten, die auf eine Collision getestet werden sollen. Wenn eine Collision vorliegt, wird im Datenregister D0 eine 1 zurückgegeben, andernfalls eine 0. Die Collsionsroutine ist unabhängig von einer BitMap. Das bedeutet, daß auch Collisionen von Objekten getestet werden, die sich außerhalb der BitMap befinden.

Routine: Collision (A0,A1) (ObjektArgs1,ObjektArgs2)
Parameter: A0 = Adresse der ObjektArgs-Struktur von BOB 1
A1 = Adresse der ObjektArgs-Struktur von BOB 2
Rückgabe: in D0 = 0, dann keine Collision
D0 = 1, dann liegt eine Collision vor
Erklärung: Diese Routine testet eine Collision zwischen zwei Objekten. Wenn eine Collision vorliegt, wird in D0 eine 1 zurückgegeben, andernfalls eine 0. Testet auch Collisionen zwischen ausgeschalteten und sichtbaren BOBs - ist nicht von einer BitMap abhängig.

Hier das dazugehörige Listing:

```
;--- Collisionsabfrage A0 = OBArgs 1; A1 = OBArgs 2;
;--- Diese Routine kann PC-Relative assembliert werden
Collision:
  move.l #$dff000,a6
  move.w 6(a1),d0          ; Ypos
  move.w 8(a1),d1          ; Xpos
  sub.w 8(a0),d1           ; relative Xpos von BOB 2
  sub.w 6(a0),d0           ; relative Ypos von BOB 2
  bmi coll_ynegativ        ; Ypos negativ ?
  cmp.w 10(a0),d0          ; If relYpos >= Höhe (BOB 1)
                             then NoColl
  bge coll_nocoll          ; Keine Collision
  bra coll_relxpos         ; relative Xpos auf Collision
                             testen
coll_ynegativ:
  move.w d0,d2             ; relYpos nach D2
  add.w 10(a1),d2          ; plus Höhe (BOB 2)
  cmp.w #0,d2              ; If relYpos+Höhe <= Null
                             then NoColl
  ble coll_nocoll          ; Keine Collision
coll_relxpos:
  tst.w d1                 ; XPos negativ ?
  bmi coll_xnegativ
  move.w 12(a0),d2         ; WordWidth (BOB 1) nach D2
```

```
  lsl.w #4,d2                ; mal 16 = Pixelbreite von
                               BOB 1
  cmp.w d2,d1                ; If relXpos >= Pixelbreite von
                               BOB1 then NoColl
  bge coll_nocoll            ; Keine Collision
  bra coll_begin             ; Collisionsberechnung be-
                               ginnen
coll_xnegativ:
  move.w 12(a1),d2           ; WordWidth (BOB 2) nach D2
  lsl.w #4,d2                ; mal 16 = Pixelbreite von
                               BOB 2
  add.w d1,d2                ; plus relXpos
  cmp.w #0,d2                ; If relXpos+Pixelbreite <=
                               Null then NoColl
  ble coll_nocoll            ; keine Collision
coll_begin:
  lea coll_blitterargs,a2    ; Blittrargs nach A2
  move.l #$ffffffff,18(a2)   ; FirstMask
  clr.w 16(a2)               ; ShiftB-Wert löschen
  clr.l d2                   ; Pos-Offset von BOB 1
  clr.l d3                   ; Pos-Offset von BOB 2
;-- ab hier Y-Offset berechnen von BOB 1 und 2 --
  tst.w d0                   ; YPos negativ ?
  bmi coll_b3
  move.w 12(a0),d4           ; WordWidth nach D4 (BOB 1)
  lsl.w #1,d4                ; mal 2 = ByteWidth
  mulu d0,d4                 ; mal relYpos von BOB 2
  add.l d4,d2                ; zu Offset BOB 1 addieren
  move.w 10(a0),d4           ; Höhe von BOB 1
  sub.w d0,d4                ; minus relYpos = HöheB1
  cmp.w 10(a1),d4            ; If HöheB1 > Höhe2 then Hö-
                               he2 = BlitterHöhe
  bgt coll_h2
  move.w d4,14(a2)           ; BlitterHöhe in Blitterargs
  bra coll_offsetx           ; X-Offset berechnen
coll_h2:
  move.w 10(a1),14(a2)       ; BlitterHöhe in Blitterargs
  bra coll_offsetx           ; X-Offset berechnen
```

```
coll_b3:                        ; ab hier negative Y-Pos
  neg.w d0                      ; relYpos jetzt positiv
  move.w 12(a1),d4              ; WordWidth nach D4
  lsl.w #1,d4                   ; mal 2 = ByteWidth
  mulu d0,d4                    ; mal relYpos = YOffset von
                                  BOB 2
  add.l d4,d3                   ; zu Offset BOB 2 addieren
  move.w 10(a1),d4              ; Höhe von BOB 2 nach D4
  sub.w d0,d4                   ; minus relYpos = HB2
  cmp.w 10(a0),d4               ; If HB2 > H1 then H1 = Blit-
                                  terHöhe
  bgt coll_h1
  move.w d4,14(a2)              ; BlitterHöhe in Blitterargs
  bra coll_offsetx              ; X-Offset berechnen
coll_h1:
  move.w 10(a0),14(a2)          ; BlitterHöhe in Blitterargs
;--- ab hier X-Offset von BOB 1 und 2 ---
coll_offsetx:
  tst.w d1                      ; relXpos negativ ?
  bmi coll_b4
  move.w d1,d4                  ; relXpos nach d4
  lsr.w #4,d4                   ; durch 16 teilen ohne Rest
  move.w 12(a0),d5              ; WordWidth von BOB 1 nach
                                  D5
  sub.w d4,d5                   ; minus WX2 = WordBreite
                                  von BOB 1
  cmp.w 12(a1),d5               ; If WB1 > W2 then W2 = Blit-
                                  terBreite
  bgt coll_w2
  move.w d5,12(a2)              ; Blitterbreite in Blitterargs
  bra coll_x2                   ; Shift- und Mask-Werte be-
                                  rechnen
coll_w2:
  move.w 12(a1),12(a2)          ; Blitterbreite in Blitterargs
coll_x2:
  lsl.w #1,d4                   ; WX2 mal 2 , damit in Bytes
  and.l #$0000ffff,d4           ; HI-Word löschen
  add.l d4,d2                   ; zu Offset BOB 1 addieren
```

```
  and.w #15,d1                ; ShiftB von relXpos ermitteln
  beq coll_blitter            ; Null ?
  move.w d1,16(a2)
  subq.b #1,d1                ; ShiftB-Wert minus 1, wegen DBRA
  move.w #$ffff,d4            ; FirstMask
coll_xloop:
  lsr.w #1,d4                 ; Maske berechnen
  dbra d1,coll_xloop
  move.w d4,18(a2)            ; FirstMask speichern
coll_b5:
  cmp.w #1,12(a2)             ; BlitterBreite = 1 ?
  bne coll_blitter
  move.w 18(a2),20(a2)        ; wenn ja, FirstMask = LastMask
  bra coll_blitter
coll_b4:                      ; ab hier negativen X-Offset berechnen
  neg.w d1                    ; relXpos jetzt positiv
  move.w d1,d4                ; relXpos nach D4
  lsr.w #4,d4                 ; durch 16 teilen ohne Rest = WX2
  move.w 12(a1),d5            ; W2 nach D5
  sub.w d4,d5                 ; minus WX2 = WB2
  cmp.w 12(a0),d5             ; If WB2 > W1 then W1 = BlitterBreite
  bgt coll_w1
  move.w d5,12(a2)            ; Blitterbreite in Blitterargs
  bra coll_x3                 ; ShiftB- und Mask-Werte berechnen
coll_w1:
  move.w 12(a0),12(a2)        ; Blitterbreite in Blitterargs
coll_x3:
  lsl.w #1,d4                 ; WX2 mal 2 = X2-Offset
  and.l #$0000ffff,d4         ; HI-Word löschen
  add.l d4,d3                 ; zu Offset von BOB 2 addieren
```

```
  and.w #15,d1                ; relXpos ; Rest ermitteln
  beq coll_blitter
  sub.l #2,d2                 ; Offset minus 2 von BOB 1
                                (d2)

  move.w #16,d4
  sub.w d1,d4                 ; ShiftB-Wert invertieren
  move.w d4,16(a2)            ; ShiftB-Wert in Blitterargs
  sub.w #1,d1
  move.w #$ffff,d7
coll_tloop:
  lsr.w #1,d7
  dbra d1,coll_tloop
  move.w d7,18(a2)
  bra coll_b5
coll_blitter:
  move.l 18(a2),$44(a6)       ; FirstMask
  move.w 12(a0),d0            ; WordWidth von BOB 1 nach
                                D0
  move.w 12(a1),d1            ; WordWidth von BOB 2 nach
                                D1
  sub.w 12(a2),d0             ; Modulo A
  sub.w 12(a2),d1             ; Modulo B
  lsl.w #1,d0                 ; Modulo A in Bytes
  lsl.w #1,d1                 ; Modulo B in Bytes
  move.w d0,$64(a6)           ; Modulo A
  move.w d1,$62(a6)           ; Modulo B
  move.w 16(a2),d0            ; ShiftB-Wert nach D0
  lsl.w #8,d0
  lsl.w #4,d0                 ; real ShiftB-Wert berechnen
  move.w d0,$42(a6)           ; Wert von BLTCON 1
  move.w #$0cc0,$40(a6)       ; BLTCON 0: USE A + B, LF7
                                + LF6
  add.l 28(a0),d2             ; CollMask-Adresse zu Offset
                                BOB 1 (A)
  add.l 28(a1),d3             ; CollMask-Adresse zu Offset
                                BOB 2 (B)
  move.l d2,$50(a6)           ; Quelle A (A0) (d2)
  move.l d3,$4c(a6)           ; Quelle B (A1) (d3)
```

```
  move.w 12(a2),d7          ; Breite in Words
  move.w 14(a2),d6          ; Höhe in Pixel
  and.w #$3ff,d6            ; Blittersize errechnen
  lsl.w #6,d6               ; D7 = breite in Words
  and.w #$3f,d7             ; D6 = höhe in Pixel
  add.w d6,d7               ; D7 ist jetzt Blittersize
  move.w d7,$58(a6)         ; Blitter starten
coll_wait:
  btst #14,$2(a6)           ; Bit BBusy testen
  bne coll_wait             ; wenn Null, dann Blitterende
  btst #13,$2(a6)           ; DMAControllregisterRead
  bne coll_nocoll
  move.l #1,d0              ; Collision stattgefunden (D0
                              = 1)
  rts
coll_nocoll:
  clr.l d0
  rts                       ; Ende
coll_blitterargs: blk.b 22,0
;--- Listingende ---
```

Zum Abschluß des Kapitels (ein Abschnitt folgt noch), ein Beispiellisting, das drei Blitterobjekte nach der dritten Methode bewegt und eine Collision zwischen zwei Objekten testet. Bei Berührung stoßen sie sich gegenseitig ab. Auf Druck der rechten Maustaste wird das Programm beendet.

Listingbeschreibung:

Auch hier schalten wir wieder das Betriebssystem aus, da wir unseren eigenen Copperinterrupt programmieren. Wir legen auch wieder eine RastPort-Struktur an, damit später auch Text ausgegeben werden kann. Nun erzeugen wir drei identische BitMap-Strukturen und printen in allen dreien einen Hintergrundtext. Ein völlig schwarzer Hintergrund wäre ja ein bißchen eintönig. Die erste BitMap wird über die Copperliste gestartet. Natürlich müssen

wir auch für alle drei Objekte die Masken (ShadowMask und CollMask) erzeugen. Damit die Funktion PrintObjekt () auch funktioniert, speichern wir die Adresse der zweiten BitMap (der unsichtbaren) in die Variable po_bitmap. Der Copperinterrupt wird nun initialisiert.

Das Verwalten von drei Objekten erfordert schon einiges an Aufwand. Deswegen legen wir uns eine BOB-Tabelle an und geben unsere BOBs über die Funktion WriteBobsOnly () alle auf einmal aus. Bevor wir zur Hauptschleife kommen, legen wir uns noch eine BitMap-Tabelle an, in der alle drei BitMap-Strukturen enthalten sind. Diese brauchen wir für das Vertauschen und Restaurieren des Hintergrundes.

Zum Hauptprogramm (Schleife):

Als erstes bewegen wir jedes einzelne Objekt und retten dabei immer die alte Position, sonst könnten wir bei einer Collision die zwei Objekte nicht voneinander abstoßen lassen. Danach werden die ersten beiden BOBs auf Collision getestet. Ist dies der Fall, so werden sie auf ihre alten Koordinaten zurückgesetzt. Beim dritten Schritt schreiben wir alle Objekte mit der Funktion WriteBobsOnly() in die unsichtbare BitMap. Die Unterroutine Change_BitMaps vertauscht die ersten beiden BitMap-Strukturen und aktiviert die unsichtbare über die Copperliste. Dabei darf nicht vergessen werden, die jetzt unsichtbare BitMap wieder in die Variable po_bitmap zu speichern. Als letzter Schritt wird die dritte BitMap mit der Funktion CopyBitMap() in die unsichtbare kopiert (Hintergrund wieder im alten Zustand).

Wenn sie auf die rechte Maustaste drücken, wird das Programm beendet.

Es folgt jetzt das dazugehörige Listing:

```
;--- Erzeugt einen Screen ü ber die Hardware      ---
;--- Printet einen Text und bewegt drei BOBs      ---
;--- im Double-Buffering (Methode 3), testet     ---
;--- eine Collision zwischen zwei BOBs            ---
;--- und auf rechte Maustaste erscheint alte      ---
;--- Copperliste und Programm wird beendet        ---
;--- Programmname = CollisionBOB                  ---
;
;--- Betriebssystem ausschalten ---
  move.l #600000,d0
motor_aus:
  sub.l #1,d0                    ; warten
  cmp.l #0,d0                    ; bis
  bne motor_aus                  ; Diskmotor aus
  move.l 4,a6                    ; IRQs aus
  jsr -120(a6)                   ; Disable ()
;
  move.l 4,a6                    ; ExecBasis
  lea gfxname,a1                 ; Libraryname
  clr.l d0                       ; Version = 0
  jsr -552(a6)                   ; OpenLibrary ()
  move.l d0,gfxbase              ; graphics.basis
;
  lea rastport,a1                ; RastPort-Struktur
  move.l gfxbase,a6              ; graphics.basis
  jsr -198(a6)                   ; InitRastPort ()
;
  lea bitmapA,a0                 ; BitMap-Struktur A
  move.l #5,d0                   ; Depth = 5
  move.l #320,d1                 ; 320 breit
  move.l #256,d2                 ; 256 hoch
  move.l #1,d3                   ; Speicher reservieren
  bsr initbitmap                 ; BitMap init.
  tst.w d0                       ; Error ?
  bne exit_1
;
```

```
  lea bitmapB,a0                ; BitMap-Struktur B
  move.l #5,d0                  ; Depth = 5
  move.l #320,d1                ; 320 breit
  move.l #256,d2                ; 256 hoch
  move.l #1,d3                  ; Speicher reservieren
  bsr initbitmap                ; BitMap init.
  tst.w d0                      ; Error ?
  bne exit_1
;
  lea bitmapC,a0                ; BitMap-Struktur C
  move.l #5,d0                  ; Depth = 5
  move.l #320,d1                ; 320 breit
  move.l #256,d2                ; 256 hoch
  move.l #1,d3                  ; Speicher reservieren
  bsr initbitmap                ; BitMap init.
  tst.w d0                      ; Error ?
  bne exit_1
;
  lea colortable,a0             ; Farbtabelle
  lea colorpuffer,a1            ; Copperpuffer
  bsr initcolor                 ; Farbregister init.
;

;--- erste BitMap  ber Copperliste aktivieren ---
  lea bitmapA,a0                ; BitMap-Struktur A
  lea copperpuffer,a1           ; Copperpuffer
  clr.l d0                      ; Modus = Normal
  bsr initcoppermap             ; Customregister init.
;
  move.l #copperlist,$dff080    ; Copperlistadresse
  clr.w $dff088                 ; Copperliste starten
;
  move.w #$20,$dff096           ; Sprite-DMA aus
  move.l #spritedatas,$dff120   ; Sprite 0 (Mauszeiger) aus
;
  lea bitmapA,a0                ; BitMap-Struktur A
  lea rastport,a1               ; RastPort-Struktur
```

```
  move.l a0,4(a1)              ; BitMap in RastPort einhängen
  lea demotext,a0              ; Textadresse
  move.l gfxbase,a6            ; graphics.basis
  clr.l d0                     ; X-Position
  move.l #100,d1               ; Y-Position
  bsr printtext                ; Text printen in BitMap A
;
  lea bitmapB,a0               ; BitMap-Struktur B
  lea rastport,a1              ; RastPort-Struktur
  move.l a0,4(a1)              ; BitMap in RastPort einhängen
  lea demotext,a0              ; Textadresse
  move.l gfxbase,a6            ; graphics.basis
  clr.l d0                     ; X-Position
  move.l #100,d1               ; Y-Position
  bsr printtext                ; Text printen in BitMap B
;
  lea bitmapC,a0               ; BitMap-Struktur C
  lea rastport,a1              ; RastPort-Struktur
  move.l a0,4(a1)              ; BitMap in RastPort einhängen
  lea demotext,a0              ; Textadresse
  move.l gfxbase,a6            ; graphics.basis
  clr.l d0                     ; X-Position
  move.l #100,d1               ; Y-Position
  bsr printtext                ; Text printen in BitMap C
;

;--- Masken initialisieren von BOB 1 ---
  lea objektargs1,a0           ; ObjektArgs-Struktur
  move.l #2,d0                 ; MEMORY + CollMask = ShadowMask
  bsr InitMask                 ; Initialisiere Masken und Puffer
  tst.w d0                     ; Error ?
  bne exit_2
;
```

```
;--- Masken initialisieren von BOB 2 ---
  lea objektargs2,a0              ; ObjektArgs-Struktur
  move.l #2,d0                    ; MEMORY + CollMask =
                                    ShadowMask
  bsr InitMask                    ; Initialisiere Masken und
                                    Puffer
  tst.w d0                        ; Error ?
  bne exit_2
;
;--- Masken initialisieren von BOB 3 ---
  lea objektargs3,a0              ; ObjektArgs-Struktur
  move.l #2,d0                    ; MEMORY + CollMask =
                                    ShadowMask
  bsr InitMask                    ; Initialisiere Masken und
                                    Puffer
  tst.w d0                        ; Error ?
  bne exit_2
;
;--- BitMap-Struktur B in Variable po_bitmap schreiben ---
  move.l #bitmapB,po_bitmap
;
;--- CopperIrq einschalten ---
  move.l $6c,oldirqebene          ; alte IRQ-Ebene retten
  move.w $dff01c,intena_buf       ; IRQ-Register retten
  move.l #CopperIrq,$6c           ; eigene IRQ-Routine
                                    einh ängen
  move.w #$7fff,$dff09a           ; alle IRQs aus
  move.w #$c010,$dff09a           ; Copper-IRQ ein
;
;--- alle Objekte in eine Tabelle eintragen ---
  lea bobtabelle,a0
  move.l a0,printbob_tabelle; in BOB-Tabelle eintragen
  move.w #3,(a0)                  ; Anzahl = 3
  move.l #objektargs1,2(a0)       ; ObjektArgs 1
  move.l #objektargs2,6(a0)       ; ObjektArgs 2
  move.l #objektargs3,10(a0)      ; ObjektArgs 3
;
```

```
;--- BitMaptabelle zum Vertauschen anlegen ---
  lea bitmap_tabelle,a0
  move.l #bitmapA,(a0)          ; sichtbare bei Offset 0
  move.l #bitmapB,4(a0)         ; unsichtbare bei Offset 4
  move.l #bitmapC,8(a0)         ; Hintergrund-Bitmap bei
                                  Offset 8
;
;--- Maustaste abfragen ---
main:
  btst #10,$dff016              ; Rechte Maustaste ?
  beq exit                      ; ja, dann Prg. ende
  bsr move_objekts
  bsr test_collision
  bsr writebobsonly             ; BOBs in unsichtbare BitMap
                                  kopieren
  bsr change_bitmaps
  bsr copy_bitmaps
  bra main                      ; zur ck zum Anfang

;--- Objekte 1 bis 3 bewegen ---
move_objekts:
  lea objektargs1,a0            ; ObjektArgs-Struktur BOB 1
  lea speed1,a1                 ; Speedpuffer BOB 1
  bsr move_bob
;
  lea objektargs2,a0
  lea speed2,a1
  bsr move_bob
;
  lea objektargs3,a0
  lea speed3,a1
  bsr move_bob
  rts
;
;--- Objekte 1 und 2 auf Collision testen ---
;--- und ggfs. gegeneinander abprallen lassen ---
test_collision:
  lea objektargs1,a0
```

```
  lea objektargs2,a1
  bsr collision                ; Collision testen
  tst.w d0                     ; liegt eine vor?
  beq test_end
  lea objektargs1,a0           ; ObjektArgs-Struktur 1
  lea speed1,a1
  move.l (a0),6(a0)            ; auf alte Position wieder set-
                                 zen
  neg.w (a1)                   ; Y-Richtung ndern
  neg.w 2(a1)                  ; X-Richtung ndern
;
  lea objektargs2,a0           ; ObjektArgs-Struktur 2
  lea speed2,a1
  move.l (a0),6(a0)            ; auf alte Position wieder set-
                                 zen
  neg.w (a1)                   ; Y-Richtung ndern
  neg.w 2(a1)                  ; X-Richtung ndern
test_end:
  rts

;--- BitMaps vertauschen und aktivieren ---
change_bitmaps:
  lea bitmap_tabelle,a3
  move.l (a3),a0               ; A0 = sichtbare BitMap
  move.l 4(a3),(a3)            ; unsichtbare nach sichtbare
  move.l a0,4(a3)              ; sichbare nach unsichtbare
  move.w #1,irq_change         ; im Copper-IRQ vertauschen
wait_irq:
  tst.w irq_change             ; auf vertauschen warten
  bne wait_irq
  move.l 4(a3),po_bitmap       ; unsichtbare BitMap in Va-
                                 riable
                               ; f r PrintObjekt ()
  rts
;
;--- Hintergrund wiederherstellen ---
copy_bitmaps:
  lea bitmap_tabelle,a3
```

```
  move.l 8(a3),a0              ; Quell-BitMap
  move.l 4(a3),a1              ; Ziel-BitMap
  bsr copybitmap               ; BitMaps kopieren
  rts
;
;--- CopperIRQ-Routine ---
copperirq:
  move.w #$0010,$dff09c        ; CoperIRQ zurü cksetzen
  movem.l d0-d7/a0-a6,-(sp)    ; Daten- Adressregister retten
;
  tst.w irq_change             ; BitMaps vertauschen ?
  beq irq_end
  move.l bitmap_tabelle,a0     ; noch unsichtbare BitMap
                               holen
  lea copperpuffer,a1          ; Copperpuffer
  clr.l d0                     ; Modus = Normal
  bsr initcoppermap            ; BitMap ber Copperliste akti-
                               vieren
  clr.w irq_change             ; Vertauschung zu ende
;
irq_end:
  movem.l (sp)+,d0-d7/a0-a6    ; Register zur ckholen
  rte                          ; IRQ-Programm beenden
;

;--- BOB nur bewegen (Positionen ndern)  ---
;--- A0 = ObjektArgs , A1 = Speed (2 Words) ---
  move_bob:
  move.l 6(a0),(a0)            ; alte Position retten
  cmp.w #288,8(a0)             ; XPos
  blt x_kleiner                ; < 288 ?
  neg.w 2(a1)                  ; Vorzeichen von X-Pos.
                               ä ndern
x_kleiner:
  cmp.w #0,8(a0)               ; XPos
  bge x_gr er_gleich
  neg.w 2(a1)                  ; Vorzeichen von X-Pos
                               ä ndern
```

```
x_grser_gleich:
  move.w 2(a1),d0              ; X-Speed
  add.w d0,8(a0)               ; New-XPos.
;
  cmp.w #246,6(a0)             ; YPos
  blt y_kleiner                ; < 246 ?
  neg.w (a1)                   ; Vorzeichen von Y-Pos.
                                 ändern
y_kleiner:
  cmp.w #0,6(a0)               ; YPos
  bge y_grser_gleich
  neg.w (a1)                   ; Vorzeichen von Y-Pos ändern
y_gr ser_gleich:
  move.w (a1),d0               ; Y-Speed
  add.w d0,6(a0)               ; New-YPos.
  rts
;
;--- Programm beenden ---
exit:
;
;--- BOBs ausschalten ---
  bsr change_bitmaps
  bsr copy_bitmaps
;
;--- CopperIRQ wieder aus ---
  or.w #$8000,intena_buf       ; Set/Clr- Bit setzen
  move.w #$7fff,$dff09a        ; alle IRQs aus
  move.w intena_buf,$dff09a    ; alte IRQs wieder einschalten
  move.l oldirqebene,$6c       ; System-IRQ-Routine einhängen
;
exit_2:
;--- von BOB 1 Maskenpuffer wieder freigeben ---
  lea objektargs1,a0           ; ObjektArgs-Struktur
  bsr freemask                 ; Maskenpuffer freigeben
;
```

```
;--- von BOB 2 Maskenpuffer wieder freigeben ---
  lea objektargs2,a0              ; ObjektArgs-Struktur
  bsr freemask                    ; Maskenpuffer freigeben

;--- von BOB 3 Maskenpuffer wieder freigeben ---
  lea objektargs3,a0              ; ObjektArgs-Struktur
  bsr freemask                    ; Maskenpuffer freigeben
;
  move.l gfxbase,a0               ; graphics.basis
  move.l 38(a0),$dff080           ; alte Copperlistadresse
  clr.w $dff088                   ; und starten
;
  move.w #$8220,$dff096           ; Sprite-DMA an
;
  lea bitmapA,a0                  ; BitMap-Struktur A
  bsr clearbitmap                 ; BitMap A freigeben
;
  lea bitmapB,a0                  ; BitMap-Struktur B
  bsr clearbitmap                 ; BitMap B freigeben
;
  lea bitmapC,a0                  ; BitMap-Struktur C
  bsr clearbitmap                 ; BitMap C freigeben
;
exit_1:
  move.l gfxbase,a1               ; graphics.basis
  move.l 4,a6                     ; exec.basis
  jsr -414(a6)                    ; CloseLibrary ()
;
;--- System wieder anschalten ---
  move.l 4,a6                     ; IRQs wieder erlauben
  jsr -126(a6)                    ; Enable ()
  rts                             ; Programmende

;--- Routine zum Copieren von Grafik mit dem Blitter          ---
;--- A0 = Quell-BitMap ; A1 = Ziel-BitMap                     ---
CopyBitMap:
  move.l #$dff000,a6
```

```
  move.l #$09f00000,$40(a6)      ; USE A und D ; Minterm: A
                                   = D
  move.l #-1,$44(a6)             ; First/Last Mask (alle Bits
                                   bernehmen)
  clr.l $64(a6)                  ; Modulowert von Quelle A
  move.w (a0),d5                 ; Breite in Bytes
  lsr.w #1,d5                    ; jetzt in Words
  move.w 2(a0),d6                ; Höhe in Pixel
  and.w #$3ff,d6                 ; Blittersize errechnen
  lsl.w #6,d6                    ; D5 = Breite in Words
  and.w #$3f,d5                  ; D6 = Höhe in Pixel
  add.w d6,d5                    ; D5 ist jetzt Blittersize
  clr.w d0
  move.b 5(a0),d0                ; Tiefe
  subq #1,d0                     ; minus 1
  add.l #8,a0
  add.l #8,a1
cbmap_loop:
  move.l (a0)+,$50(a6)           ; Anfangsadresse von Quelle
                                   A
  move.l (a1)+,$54(a6)           ; Anfangsadresse von Ziel D
  move.w d5,$58(a6)              ; BLTSIZE und Blitteroperati-
                                   on starten
cbmap_wait:
  btst #14,$2(a6)                ; Warten bis Blit fertig
  bne cbmap_wait
  dbra d0,cbmap_loop
  rts

;--- Nur BOB-Imagedaten in BitMap kopieren ---
;--- printbob_tabelle = BOB-Tabelle ---
WriteBobsOnly:
  move.l printbob_tabelle(pc),a1
  move.w (a1)+,d0                ; Anzahl Einträge
  tst.w d0
  beq writebobsonly_end
  clr.l d1
  move.w d0,d1
```

```
  lsl.w #2,d1                    ; Anzahl mal 4
  add.l d1,a1                    ; Zeiger auf letzten Eintrag
  subq.w #1,d0                   ; Anzahl minus 1
wb_loop:
  move.l -(a1),a0                ; Zeiger auf ObjektArgs
  cmp.l #0,a0                    ; überhaupt vorhanden ?
  beq wb_loop1
  move.b #2,4(a0)                ; Write BOB only
  movem.l d0/a0-a1,-(sp)
  bsr printobjekt          ;
  movem.l (sp)+,d0/a0-a1
wb_loop1:
  dbra d0,wb_loop
writebobsonly_end:
  rts

printbob_tabelle: dc.l 0

;---- Collisionsabfrage A0 = OBArgs 1; A1 = OBArgs 2; ----
;---- Diese Routine kann PC-Relative assembliert werden ---
collision:
  move.l #$dff000,a6
  move.w 6(a1),d0                ; Ypos
  move.w 8(a1),d1                ; Xpos
  sub.w 8(a0),d1                 ; relative Xpos von BOB 2
  sub.w 6(a0),d0                 ; relative Ypos von BOB 2
  bmi coll_ynegativ              ; Ypos negativ ?
  cmp.w 10(a0),d0                ; If relYpos >= H he (BOB 1)
                                   then NoColl
  bge coll_nocoll                ; Keine Collision
  bra coll_relxpos               ; relative Xpos auf Collision
                                   testen
coll_ynegativ:
  move.w d0,d2                   ; relYpos nach D2
  add.w 10(a1),d2                ; plus H he (BOB 2)
  cmp.w #0,d2                    ; If relYpos+H he <= Null then
                                   NoColl
  ble coll_nocoll                ; Keine Collision
```

```
coll_relxpos:
  tst.w d1                    ; XPos negativ ?
  bmi coll_xnegativ
  move.w 12(a0),d2            ; WordWidth (BOB 1) nach D2
  lsl.w #4,d2                 ; mal 16 = Pixelbreite von
                                BOB 1
  cmp.w d2,d1                 ; If relXpos >= Pixelbreite von
                                BOB1 then NoColl
  bge coll_nocoll             ; Keine Collision
  bra coll_begin              ; Collisionsberechnung be-
                                ginnen
coll_xnegativ:
  move.w 12(a1),d2            ; WordWidth (BOB 2) nach D2
  lsl.w #4,d2                 ; mal 16 = Pixelbreite von
                                BOB 2
  add.w d1,d2                 ; plus relXpos
  cmp.w #0,d2                 ; If relXpos+Pixelbreite <=
                                Null then NoColl
  ble coll_nocoll             ; keine Collision
coll_begin:
  lea coll_blitterargs,a2     ; Blitterargs nach A2
  move.l #$ffffffff,18(a2)    ; FirstMask
  clr.w 16(a2)                ; ShiftB-Wert l schen
  clr.l d2                    ; Pos-Offset von BOB 1
  clr.l d3                    ; Pos-Offset von BOB 2
;-- ab hier Y-Offset berechnen von BOB 1 und 2 --
  tst.w d0                    ; YPos negativ ?
  bmi coll_b3
  move.w 12(a0),d4            ; WordWidth nach D4 (BOB 1)
  lsl.w #1,d4                 ; mal 2 = ByteWidth
  mulu d0,d4                  ; mal relYpos von BOB 2
  add.l d4,d2                 ; zu Offset BOB 1 addieren
  move.w 10(a0),d4            ; Höhe von BOB 1
  sub.w d0,d4                 ; minus relYpos = H heB1
  cmp.w 10(a1),d4             ; If H heB1 > H he2 then H he2
                                = BlitterH he
  bgt coll_h2
  move.w d4,14(a2)            ; BlitterHöhe in Blitterargs
```

```
  bra coll_offsetx              ; X-Offset berechnen
coll_h2:
  move.w 10(a1),14(a2)          ; BlitterHöhe in Blitterargs
  bra coll_offsetx              ; X-Offset berechnen
coll_b3:                        ; ab hier negative Y-Pos
  neg.w d0                      ; relYpos jetzt positiv
  move.w 12(a1),d4              ; WordWidth nach D4
  lsl.w #1,d4                   ; mal 2 = ByteWidth
  mulu d0,d4                    ; mal relYpos = YOffset von
                                  BOB 2
  add.l d4,d3                   ; zu Offset BOB 2 addieren
  move.w 10(a1),d4              ; Höhe von BOB 2 nach D4
  sub.w d0,d4                   ; minus relYpos = HB2
  cmp.w 10(a0),d4               ; If HB2 > H1 then H1 = Blit-
                                  terHöhe
  bgt coll_h1
  move.w d4,14(a2)              ; BlitterH he in Blitterargs
  bra coll_offsetx              ; X-Offset berechnen
coll_h1:
  move.w 10(a0),14(a2)          ; BlitterH he in Blitterargs
;--- ab hier X-Offset von BOB 1 und 2 ---
coll_offsetx:
  tst.w d1                      ; relXpos negativ ?
  bmi coll_b4
  move.w d1,d4                  ; relXpos nach d4
  lsr.w #4,d4                   ; durch 16 teilen ohne Rest
  move.w 12(a0),d5              ; WordWidth von BOB 1 nach
                                  D5
  sub.w d4,d5                   ; minus WX2 = WordBreite
                                  von BOB 1
  cmp.w 12(a1),d5               ; If WB1 > W2 then W2 = Blit-
                                  terBreite
  bgt coll_w2
  move.w d5,12(a2)              ; Blitterbreite in Blitterargs
  bra coll_x2                   ; Shift- und Mask-Werte be-
                                  rechnen
coll_w2:
  move.w 12(a1),12(a2)          ; Blitterbreite in Blitterargs
```

```
coll_x2:
  lsl.w #1,d4                ; WX2 mal 2 , damit in Bytes
  and.l #$0000ffff,d4        ; HI-Word löschen
  add.l d4,d2                ; zu Offset BOB 1 addieren
  and.w #15,d1               ; ShiftB von relXpos ermitteln

  beq coll_blitter           ; Null ?
  move.w d1,16(a2)
  subq.b #1,d1               ; ShiftB-Wert minus 1, wegen DBRA

  move.w #$ffff,d4           ; FirstMask
coll_xloop:
  lsr.w #1,d4                ; Maske berechnen
  dbra d1,coll_xloop
  move.w d4,18(a2)           ; FirstMask speichern
coll_b5:
  cmp.w #1,12(a2)            ; BlitterBreite = 1 ?
  bne coll_blitter
  move.w 18(a2),20(a2)       ; wenn ja, FirstMask = LastMask

  bra coll_blitter
coll_b4:                     ; ab hier negativen X-Offset berechnen

  neg.w d1                   ; relXpos jetzt positiv
  move.w d1,d4               ; relXpos nach D4
  lsr.w #4,d4                ; durch 16 teilen ohne Rest = WX2

  move.w 12(a1),d5           ; W2 nach D5
  sub.w d4,d5                ; minus WX2 = WB2
  cmp.w 12(a0),d5            ; If WB2 > W1 then W1 = BlitterBreite

  bgt coll_w1
  move.w d5,12(a2)           ; Blitterbreite in Blitterargs
  bra coll_x3                ; ShiftB- und Mask-Werte berechnen

coll_w1:
  move.w 12(a0),12(a2)       ; Blitterbreite in Blitterargs
```

```
coll_x3:
  lsl.w #1,d4                    ; WX2 mal 2 = X2-Offset
  and.l #$0000ffff,d4            ; HI-Word löschen
  add.l d4,d3                    ; zu Offset von BOB 2 addie-
                                   ren
  and.w #15,d1                   ; relXpos ; Rest ermitteln
  beq coll_blitter
  sub.l #2,d2                    ; Offset minus 2 von BOB 1
                                   (d2)
  move.w #16,d4
  sub.w d1,d4                    ; ShiftB-Wert invertieren
  move.w d4,16(a2)               ; ShiftB-Wert in Blitterargs
  sub.w #1,d1
  move.w #$ffff,d7
coll_tloop:
  lsr.w #1,d7
  dbra d1,coll_tloop
  move.w d7,18(a2)
  bra coll_b5
coll_blitter:
  move.l 18(a2),$44(a6)          ; FirstMask
  move.w 12(a0),d0               ; WordWidth von BOB 1 nach
                                   D0
  move.w 12(a1),d1               ; WordWidth von BOB 2 nach
                                   D1
  sub.w 12(a2),d0                ; Modulo A
  sub.w 12(a2),d1                ; Modulo B
  lsl.w #1,d0                    ; Modulo A in Bytes
  lsl.w #1,d1                    ; Modulo B in Bytes
  move.w d0,$64(a6)              ; Modulo A
  move.w d1,$62(a6)              ; Modulo B
  move.w 16(a2),d0               ; ShiftB-Wert nach D0
  lsl.w #8,d0
  lsl.w #4,d0                    ; real ShiftB-Wert berechnen
  move.w d0,$42(a6)              ; Wert von BLTCON 1
  move.w #$0cc0,$40(a6)          ; BLTCON 0: USE A + B, LF7
                                   + LF6
```

```
  add.l 28(a0),d2                ; CollMask-Adresse zu Offset
                                   BOB 1 (A)
  add.l 28(a1),d3                ; CollMask-Adresse zu Offset
                                   BOB 2 (B)
  move.l d2,$50(a6)              ; Quelle A (A0) (d2)
  move.l d3,$4c(a6)              ; Quelle B (A1) (d3)
  move.w 12(a2),d7               ; Breite in Words
  move.w 14(a2),d6               ; Höhe in Pixel
  and.w #$3ff,d6                 ; Blittersize errechnen
  lsl.w #6,d6                    ; D7 = breite in Words
  and.w #$3f,d7                  ; D6 = Höhe in Pixel
  add.w d6,d7                    ; D7 ist jetzt Blittersize
  move.w d7,$58(a6)              ; Blitter starten
coll_wait:
  btst #14,$2(a6)                ; Bit BBusy testen
  bne coll_wait                  ; wenn Null, dann Blitterende
  btst #13,$2(a6)                ; DMAControllregisterRead
  bne coll_nocoll
  move.l #1,d0                   ; Collision stattgefunden (D0
                                   = 1)
  rts
coll_nocoll:
  clr.l d0
  rts                            ; Ende.
coll_blitterargs: blk.b 22,0

;--- ShadowMask-Puffer wieder freigeben --
;-- A0 = ObjektArgs --
FreeMask:
  move.l 20(a0),a1               ; ShadowMask-Zeiger
  cmp.l #0,a1                    ; vorhanden ?
  bne fm_1
  rts
fm_1:
  clr.l 20(a0)                   ; ShadowMask-Zeiger löschen
  move.w 12(a0),d0               ; WordWidth
  lsl.w #1,d0                    ; mal 2 = ByteWidth
```

```
  mulu 10(a0),d0                ; mal Höhe = ShadowMask-
                                  Size
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                  ; FreeMem
  rts

;---- ShadowMask anlegen
;--- A0 = ObjektArgs, D0 = Memory/CollMask-Zeiger
InitMask:
  tst.w d0                      ; Muß ShadowMask-Puffer re-
                                  serviert werden ?
  beq im_1
  movem.l d0,-(sp)
  move.w 12(a0),d0              ; WordWidth
  lsl.w #1,d0                   ; mal 2 = Bytes
  mulu 10(a0),d0                ; mal Höhe = ShadowMask-
                                  Size
  move.l #$10002,d1             ; Chip und ClearMem
  move.l #$4,a6
  move.l (a6),a6
  movem.l a0,-(sp)
  jsr -198(a6)                  ; AllocMem
  movem.l (sp)+,a0
  tst.l d0
  bne im_1a
  movem.l (sp)+,d0
  move.l #-1,d0                 ; Error
  rts
im_1a:
  move.l d0,20(a0)              ; ShadowMask-Adresse spei-
                                  chern
  movem.l (sp)+,d0
  cmp.w #2,d0
  bne im_1
  move.l 20(a0),28(a0)
im_1:
  move.l #$dff000,a6
```

```
  move.l #$0dfc0000,$40(a6)     ; BLTCON0: A + B = D
  move.l #-1,$44(a6)            ; First/Last Mask
  clr.l $62(a6)                 ; Modulowert von Quelle B
  clr.w $66(a6)                 ; Modulowert von Ziel  D
  move.w 12(a0),d5              ; Breite in Words
  move.w 10(a0),d6              ; Höhe in Pixel
  move.w d5,d0                  ; Breite nach D0
  lsl.w #1,d0                   ; mal 2 = Breite in Bytes
  mulu d6,d0                    ; D0 = ImageMapSize
  and.w #$3ff,d6                ; Blittersize errechnen
  lsl.w #6,d6                   ; D5 = breite in Words
  and.w #$3f,d5                 ; D6 = Höhe in Pixel
  add.w d6,d5                   ; D5 ist jetzt Blittersize
  move.w 14(a0),d1              ; Anzahl Planes nach D1
  sub.w #1,d1                   ; minus 1, wegen DBra
  move.l 20(a0),a1              ; A1 = ShadowMask (Ziel D
                                  und Quelle B)
  move.l 16(a0),a2              ; A2 = Image-Zeiger
initmask_loop:
  move.l a2,$50(a6)             ; Anfangsadresse von Quelle
                                  A
  move.l a1,$54(a6)             ; Anfangsadresse von Ziel  D
  move.l a1,$4c(a6)             ; Anfangsadresse von Quelle
                                  B = Ziel D
  move.w d5,$58(a6)             ; BLTSIZE und Blitteroperati-
                                  on starten
initmask_wait:
  btst #14,$2(a6)               ; Blitter fertig?
  bne initmask_wait
  add.l d0,a2                   ; nä chste ImageMap
  dbra d1,initmask_loop
  clr.l d0                      ; No Errors
  rts
```

```
;---- ein Objekt auf Bildschirm printen
;---- A0 = Objektargs
;---- po_bitmap = Zeiger auf Bitmap-Struktur
;---- Objekte nicht höher und/oder breiter als Bitmap
;---- Diese Routine kann PC-Relative assembliert werden
PrintObjekt:
  move.l po_bitmap(pc),a1
  move.l #$dff000,a2            ; Chip-Basisadresse $DFF000
  move.w (a1),d6
  lsr.w #1,d6                   ; D6 = Screenbreite in Words
  move.w 2(a1),d7               ; D7 = Screenhöhe
;
  move.w 6(a0),d0               ; Ypos nach D0
  cmp.w 2(a1),d0                ; If YPos >= Screenhöhe then
                                  ende
  bge po_clipping
  move.w 10(a0),d1              ; Höhe des Objekts nach D1
  neg.w d1                      ; negativ machen
  cmp.w d1,d0                   ; If YPos <= D1 then ende
  ble po_clipping
  move.w 8(a0),d0               ; X-Pos. nach D0
  move.w (a1),d1                ; Zeilenbreite in Bytes
  lsl.w #3,d1                   ; mal 8 = Zeilenbreite in Pixel
  cmp.w d1,d0                   ; If XPos >= Zeilenbreite then
                                  ende
  bge po_clipping
  move.w 12(a0),d1              ; Objektbreite nach D1
  sub.w #1,d1                   ; ein Word abziehen
  lsl.w #4,d1                   ; mal 16 = Breite in Pixel
  neg.w d1                      ; Objektbreite jetzt negativ
  cmp.w d1,d0                   ; If XPos <= D1 then ende
  ble po_clipping
;
  cmp.b #1,4(a0)                ; BOBOff ?
  bne po_bobon
po_bob_aus:
  bsr po_writehintergrund
  clr.b 5(a0)                   ; BOB nicht mehr init.
```

```
  rts
po_bobon:
  tst.b 4(a0)                    ; Objekt normal anschalten ?
  bne po_bobon_xa
  bsr po_writehintergrund
  bsr po_readhintergrund         ; Hintergrund speichern
  bsr po_writeobjekt             ; Objekt in Hintergrund prin-
                                   ten
  rts                            ; Ende.
po_bobon_xa:
  cmp.b #2,4(a0)                 ; Write BOB only
  bne po_bobon_xb
  bsr po_writeobjekt             ; Hintergrund zurückschrei-
                                   ben
  rts
po_bobon_xb:
  cmp.b #3,4(a0)                 ; Write Hintergrund only
  bne po_bobon_xc
  bsr po_writehintergrund
  rts
po_bobon_xc:
  cmp.b #4,4(a0)                 ; Read Hintergrund only
  bne po_bobon_end
  bsr po_readhintergrund
po_bobon_end:
  rts
po_clipping:
  tst.b 4(a0)                    ; Objekt normal anschalten ?
  beq po_bob_aus
  cmp.b #1,4(a0)
  beq po_bob_aus
  cmp.b #3,4(a0)
  beq po_bob_aus
  rts
```

```
;--- Masken/Offset/Blittersize berechnen                    ---
;--- D0 = Y, D1 = X Position                                ---
;--- Rückgabe: D0 = Blitterhöhe, D1 = Blitterbreite         ---
;         D2 = PositionOffset, D5 = Shiftwert               ---
;         D4 = muß zum Bitmapoffset addiert werden
po_parameter:
  tst.w d0                      ; Y positiv ?
  bpl po_para_1                 ; wenn ja, dann verzweigen
  move.w 12(a0),d2              ; Breite Objekt Words
  lsl.w #1,d2                   ; in Bytes
  move.w d0,d3
  neg.w d3
  mulu d3,d2
  add.w 10(a0),d0               ; plus Höhe Objekt
  bra po_para_x
po_para_1:
  move.w d7,d2                  ; ScreenHöhe
  sub.w d0,d2                   ; minus y-pos
  move.w d2,d0                  ; D0 = ergebnis
  clr.l d2
  cmp.w 10(a0),d0               ; minus Höhe
  bmi po_para_x                 ; negativ ?
  move.w 10(a0),d0              ; wenn positiv dann normale
                                  Height
po_para_x:                      ; D0 = Blitterhöhe , D2 = Y-
                                  Offset
  tst.w d1                      ; X positiv ?
  bpl po_para_2                 ; wenn ja, dann verzweigen
  neg.w d1                      ; X-pos jetzt positiv
  move.w d1,d4                  ; x-pos nach D4
  lsr.w #4,d1                   ; durch 16 teilen
  move.w 12(a0),d5              ; Objektbreite nach D5
  sub.w d1,d5
  clr.l d3
  move.w d1,d3
  lsl.w #1,d3                   ; D3 = X-Offset
  add.l d3,d2                   ; D2 = Position-Offset
  move.w d5,d1                  ; D1 = Blitterbreite
```

```
  and.w #15,d4
  clr.w d5
  tst.w d4
  beq po_para_44
  move.w #16,d5
  sub.w d4,d5                    ; D5 = real Shiftwert
  subq.w #1,d4
  move.w #$ffff,d3
po_para_shift:
  lsr.w #1,d3
  dbra d4,po_para_shift
  move.w d3,$44(a2)              ; FirstMask
  move.w d3,$46(a2)
  cmp.w #1,d1
  beq po_para_7
  move.w #$ffff,$46(a2)          ; LastMask
po_para_7:
  move.l #2,d4                   ; Bitmapoffset -2 Bytes
  rts
po_para_2:                       ; X-Pos ist positiv
  move.w d1,d5                   ; X-pos nach D5
  and.w #15,d5                   ; D5 = Shiftwert
  lsr.w #4,d1                    ; X-Pos durch 16
  move.w d6,d4                   ; Screenbreite nach D4
  sub.w d1,d4
  move.w d4,d1                   ; D1 = Blitterbreite
  cmp.w 12(a0),d1                ; Ergebnis - Objektbreite
  bmi po_para_3                  ; hiernach normal weiter
  move.w 12(a0),d1               ; Blitterbreite
po_para_44:
  move.l #-1,$44(a2)             ; First/LastMask
  clr.l d4                       ; Bitmapoffset + 0 Bytes
  rts
po_para_3:
  tst.w d5
  beq po_para_44
  move.w d5,d3
  subq.w #1,d3
```

```
  move.w #$ffff,d4
po_para_shifta:
  lsl.w #1,d4
  dbra d3,po_para_shifta
  move.w d4,$44(a2)              ; FirstMask
  move.w d4,$46(a2)
  cmp.w #1,d1                    ; Blitterbreite = 1
  beq po_para_4
  move.w #$ffff,$44(a2)          ; LastMask
po_para_4:
  clr.l d4                       ; Bitmapoffset + 0 Bytes
  rts


;----- Hintergrund wieder printen ----
po_writehintergrund:
  tst.b 5(a0)                    ; wurde ein Hintergrund
                                   schon gelesen ?
  bne po_writehg_x
  rts
po_writehg_x:
  move.w (a0),d0                 ; OldY
  move.w 2(a0),d1                ; OldX
  bsr po_parameter
  move.l #-1,$44(a2)             ; First/LastMask
  move.l #$09f00000,$40(a2)      ; BLTCON0/1
  move.l 24(a0),a3               ; A3 = real Quelle A
  add.l d2,a3
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)              ; ZModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)              ; AModulo
```

```
  clr.l d3
  tst.w 2(a0)
  bmi po_writehg_nox
  move.w 2(a0),d3              ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                  ; X-Offset
po_writehg_nox:
  tst.w (a0)
  bmi po_writehg_noy
  move.w (a0),d4
  mulu d6,d4
  lsl.l #1,d4                  ; Y-Offset
  add.l d4,d3                  ; D3 = Bitmapoffset
po_writehg_noy:
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                  ; D4 = Map-Size vom Objekt
  clr.w d5
  move.b 5(a1),d5
  subq.w #1,d5                 ; D5 = Loop-z hler
  and.w #$3ff,d0               ; Blittersize errechnen
  lsl.w #6,d0                  ; D1 = Breite in Words
  and.w #$3f,d1                ; D0 = Höhe in Pixel
  add.w d0,d1                  ; D1 ist jetzt Blittersize
po_writehg_loop:
  move.l a3,$50(a2)            ; Quelle A
  move.l (a4)+,a5
  add.l d3,a5
  move.l a5,$54(a2)            ; Ziel D
  move.w d1,$58(a2)            ; Blitter starten
po_writehg_wait:
  btst #14,$2(a2)              ; Bit BBusy testen
  bne po_writehg_wait          ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_writehg_loop
  rts
```

```
;------ Hintergrund speichern --------
po_readhintergrund:
  move.b #1,5(a0)              ;    Init = 1, Hintergrund schon
                                    mal gelesen
  move.w 6(a0),0(a0)           ; Ypos nach OldYpos
  move.w 8(a0),2(a0)           ; Xpos nach OldXpos
  move.w (a0),d0               ; Y
  move.w 2(a0),d1              ; X
  bsr po_parameter
  move.l #-1,$44(a2)           ; First/LastMask
  move.l #$09f00000,$40(a2)  ; BLTCON0/1
  move.l 24(a0),a3             ; A3 = Ziel D
  add.l d2,a3
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)                ; AModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)                ; ZModulo
  clr.l d3
  tst.w 2(a0)
  bmi po_readhg_nox
  move.w 2(a0),d3                  ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                      ; X-Offset
po_readhg_nox:
  tst.w (a0)
  bmi po_readhg_noy
  move.w (a0),d4
  mulu d6,d4
  lsl.l #1,d4                      ; Y-Offset
  add.l d4,d3                      ; D3 = Bitmapoffset
po_readhg_noy:
  move.w 12(a0),d4
  mulu 10(a0),d4
```

```
  lsl.l #1,d4                    ; D4 = Map-Size vom Objekt
  clr.w d5
  move.b 5(a1),d5
  subq.w #1,d5                   ; D5 = Loop-z hler
  and.w #$3ff,d0                 ; Blittersize errechnen
  lsl.w #6,d0                    ; D1 = Breite in Words
  and.w #$3f,d1                  ; D0 = Höhe in Pixel
  add.w d0,d1                    ; D1 ist jetzt Blittersize
po_readhg_loop:
  move.l a3,$54(a2)              ; Ziel D
  move.l (a4)+,a5
  add.l d3,a5
  move.l a5,$50(a2)              ; Quelle A
  move.w d1,$58(a2)              ; Blitter starten
po_readhg_wait:
  btst #14,$2(a2)                ; Bit BBusy testen
  bne po_readhg_wait             ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_readhg_loop
  rts


;--- Objekt Daten in Bitmap kopieren ---
po_writeobjekt:
  move.w 6(a0),d0                ; Y
  move.w 8(a0),d1                ; X
  bsr po_parameter
  lsl.w #8,d5
  lsl.w #4,d5                    ; korrekter Shiftwert
  move.w d5,$42(a2)              ; BLTCON1
  add.w #$0fca,d5
  movem.l d5,-(sp)
  move.w d5,$40(a2)              ; BLTCON0
  move.l 20(a0),a5
  add.l d2,a5                    ; A5 = Quelle A (real Shadow-
                                 Mask)
  move.l 16(a0),a3
  add.l d2,a3                    ; A3 = Quelle B (real Image)
```

```
  lea 8(a1),a4
  move.w d6,d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$66(a2)          ; ZModulo
  move.w d5,$60(a2)          ; CModulo
  move.w 12(a0),d5
  sub.w d1,d5
  lsl.w #1,d5
  move.w d5,$64(a2)          ; AModulo
  move.w d5,$62(a2)          ; BModulo
  clr.l d3
  tst.w 8(a0)
  bmi po_writeo_nox
  move.w 8(a0),d3            ; x-pos
  lsr.w #4,d3
  lsl.w #1,d3                ; X-Offset
po_writeo_nox:
  tst.w 6(a0)
  bmi po_writeo_noy
  move.w 6(a0),d5
  mulu d6,d5
  lsl.l #1,d5                ; Y-Offset
  add.l d5,d3
po_writeo_noy:
  sub.l d4,d3                ; D3 = Bitmapoffset
  move.w 12(a0),d4
  mulu 10(a0),d4
  lsl.l #1,d4                ; D4 = Map-Size vom Objekt
  move.w 14(a0),d5
  subq.w #1,d5               ; D5 = Loop-zähler
  and.w #$3ff,d0
  and.w #$3f,d1
  lsl.w #6,d0                ; D1 = breite in Words
  add.w d0,d1                ; D1 ist jetzt Blittersize
po_writeo_loop:
  move.l a5,$50(a2)          ; Quelle A (ShadowMask)
  move.l a3,$4c(a2)          ; Quelle B (Image)
```

```
  move.l (a4)+,a6
  add.l d3,a6
  move.l a6,$48(a2)          ; Quelle C (BitMap)
  move.l a6,$54(a2)          ; Ziel  D (BitMap)
  move.w d1,$58(a2)          ; Blitter starten

po_writeo_wait:
  btst #14,$2(a2)            ; Bit BBusy testen
  bne po_writeo_wait         ; wenn Null, dann Blitterende
  add.l d4,a3
  dbra d5,po_writeo_loop
  movem.l (sp)+,d5           ; BLTCON0 wert holen
  move.w 14(a0),d0           ; Depth BOB nach D0
  cmp.b 5(a1),d0             ; = Anzahl Planes Screen ?
  beq po_writeo_end          ; Wenn ja, dann Ende
  sub.b 5(a1),d0             ; Planes-Anzahl abziehen
  neg.b d0                   ; Positiv machen
  subq.w #1,d0
  sub.w #$0400,d5            ; DMA-Kanal B = aus
  move.w d5,$40(a2)          ; BLTCON0
  clr.w $42(a2)              ; No Shift B
  clr.w $72(a2)              ; Clear Datenregister B (Figur)

po_writeo_loop2:
  move.l a5,$50(a2)          ; Quelle A (ShadowMask)
  move.l (a4)+,a6
  add.l d3,a6
  move.l a6,$48(a2)          ; Quelle C (BitMap)
  move.l a6,$54(a2)          ; Ziel  D (BitMap)
  move.w d1,$58(a2)          ; Blitter starten
po_writeo_wait2:
  btst #14,$2(a2)            ; Bit BBusy testen
  bne po_writeo_wait2        ; wenn Null, dann Blitterende
  dbra d0,po_writeo_loop2
po_writeo_end:
  rts
po_bitmap: dc.l 0
```

```
;--- Gibt einen Text auf dem Bildschirm aus,
;--- der mit einem Nullbyte endet
;--- Parameter: A0 = Text, A1 = RastPort,
;---            A6 = Graphics-Basis
;---            D0 = X-Pos., D1 = Y-Position
PrintText:
  clr.w d2                          ; Zähler für Anzahl Zeichen
  move.l a0,a2
pt_loop:
  tst.b (a2)+
  beq pt_loop_end
  add.w #1,d2
  cmp.w #100,d2                     ; Max. 80 Zeichen
  bne pt_loop
pt_loop_end:
  tst.w d2                          ; Kein Text ?
  beq pt_end
  movem.l a0-a1/a6/d2,-(sp)         ; Parameter retten
  jsr -240(a6)                      ; MOVE () - Position setzen
  movem.l (sp)+,a0-a1/a6/d0         ; Parameter holen
  jsr -60(a6)                       ; TEXT () - Text printen
pt_end:
  rts


;--- BitMap-Pointer in CopperList bertragen ---
;--- A0 = BitMap ; A1 = CopperPuffer ; D0 = Modus ---
InitCopperMap:
  move.w (a0),d1                    ; Bytes pro Zeile nach D1
  sub.w #40,d1                      ; minus 40 Bytes (320 Pixel)
  cmp.w #$8000,d0                   ; Modus = Hires ?
  bne icm_1
  sub.w #40,d1                      ; nochmal minus 40 Bytes
                                      (insgesamt -640 Pixel)
icm_1:
  cmp.w #$04,d0                     ; Modus = Interlace ?
  bne icm_2
```

```
  sub.w #40,d1                ; auch minus 40 Bytes (insgesammt -640 Pixel)
icm_2:
  move.w #$0108,(a1)+         ; BPL1MOD
  move.w d1,(a1)+             ; Modulo-Wert ungerade Planes
  move.w #$010a,(a1)+         ; BPL2MOD
  move.w d1,(a1)+             ; Modulo-Wert gerade Planes
  move.b 5(a0),d1             ; Depth nach D1
  lsl.w #8,d1                 ; Korrekten
  lsl.w #5,d1                 ; Depth-Wert
  lsr.w #1,d1                 ; ermitteln
  add.w #$0200,d1             ; Color setzen
  add.w d0,d1                 ; Modus setzen
  move.w #$0100,(a1)+         ; BPLCON0
  move.w d1,(a1)+             ; setzen
  moveq #5,d0                 ; max. Tiefe minus 1
  move.w #$00e0,d1            ; Hi-Word vom ersten Map-Pointer
  add.l #8,a0                 ; Erster Pointer-Zeiger
icm_loop:
  move.w d1,(a1)+             ; HI-Word vom Pointer
  move.w (a0)+,(a1)+          ; setzen
  add.w #2,d1                 ; Lo-Word ermitteln
  move.w d1,(a1)+             ; und
  move.w (a0)+,(a1)+          ; setzen
  add.w #2,d1                 ; Hi-Word ermitteln
  dbra d0,icm_loop
  rts

;---- ColorMap in CopperListe bertragen ---
;--- A0 = ColorTable ; A1 = CopperPuffer ---
InitColor:
  move.w #$180,d1             ; erstes Farbregister
  move.w #31,d0               ; Anzahl Farben
ic_loop:
  move.w d1,(a1)+             ; Farbregister in CopperList
```

```
  move.w (a0)+,(a1)+          ; dann der Farbwert in Copper-
                                List
  add.w #2,d1                 ; nächstes Farbregister
  dbra d0,ic_loop
  rts

;--- BitMap-Struktur initialisieren und
;--- Speicher f r Maps reserverieren
;--- D0 = Depth, D1 = Width, D2 = Height, D3 = Memory,
;--- A0 = BitMap
InitBitMap:
  move.w d1,d4                ; Breite nach D4
  and.w #15,d4                ; Rest ermitteln
  tst.w d4                    ; Rest vorhanden ?
  beq ibm_1                   ; Wenn nicht, dann weiter -
                                sonst Error
  move.l #-1,d0               ; Error: Breite nicht durch 16
                                teilbar
  rts
ibm_1:
  lsr.w #4,d1                 ; Breite durch 16 teilen
  lsl.w #1,d1                 ; mal 2 = Anzahl Bytes einer
                                Zeile
  move.w d1,(a0)              ; und in BitMap-Struktur
                                speichern
  move.w d2,2(a0)             ; Höhe in BitMap-Struktur
                                speichern
  move.w d0,4(a0)             ; Anzahl Planes speichern
  tst.w d3                    ; Muß Speicher f r BitMaps re-
                                serviert werden ?
  bne ibm_2                   ; Wenn nicht, dann Ende,
                                sonst weiter
  clr.l d0                    ; No Errors
  rts                         ; Ende
ibm_2:
  move.w d0,d4                ; Depth nach D4
  sub.w #1,d4                 ; minus 1
  lea 8(a0),a1                ; BitMap nach a1
```

```
ibm_clear_loop:                 ; Pointer-Zeiger
  clr.l (a1)+                   ; löschen
  dbra d4,ibm_clear_loop
  mulu d2,d1                    ; BytePerRow x Höhe = Size
                                  von einer Map
  move.w d0,d2                  ; D2 = Anzahl Planes
  move.l d1,d0                  ; D0 = ByteSize
  move.l #$10002,d1             ; D1 = CHIP + FreeMem
  sub.w #1,d2                   ; Anzahl Planes minus 1, we-
                                  gen DBRA
  add.l #8,a0                   ; Anfang BitMap-Zeiger
  move.l a0,a2                  ; auch nach A2
ibm_loop:
  move.l #$4,a6
  move.l (a6),a6
  movem.l d0-d2/a0-a2,-(sp)
  jsr -198(a6)                  ; AllocMem
  tst.l d0                      ; konnte Speicher reserviert
                                  werden ?
  bne ibm_l1                    ; Wenn ja, dann weiter
  movem.l (sp)+,d0-d2/a0-a2
ibm_free_loop:
  move.l (a2),a1                ; Zeiger auf BitMap holen
  cmp.l #0,a1                   ; Null ?
  bne ibm_l2                    ; wenn ja, dann ende
  move.l #-2,d0                 ; Error
  rts                           ; Ende
ibm_l2:
  clr.l (a2)+
  movem.l d0/a2,-(sp)
  move.l #$4,a6
  move.l (a6),a6
  jsr -210(a6)                  ; FreeMem
  movem.l (sp)+,d0/a2
  bra ibm_free_loop
ibm_l1:
  move.l d0,d5                  ; Zeiger auf BitMap nach D5
  movem.l (sp)+,d0-d2/a0-a2
```

```
  move.l d5,(a0)+
  dbra d2,ibm_loop
  clr.l d0
  rts

;--- Speicher der Maps wieder freigeben
;--- in einer BitMap-Struktur
;--- A0 = BitMap
ClearBitMap:
  clr.w d1
  move.b 5(a0),d1                ; Depth nach D1
  sub.w #1,d1
  move.w (a0),d0                 ; BytePerRow
  mulu 2(a0),d0                  ; mal Höhe = MapSize
  add.l #8,a0
  move.l #$4,a6
  move.l (a6),a6
cbm_loop:
  move.l (a0),a1                 ; Map-Pointer
  cmp.l #0,a1
  beq cbm_l1
  clr.l (a0)+
  movem.l d0-d1/a0/a6,-(sp)
  jsr -210(a6)                   ; FreeMem
  movem.l (sp)+,d0-d1/a0/a6
cbm_l1:
  dbra d1,cbm_loop
  rts

;--- Parameter ---
;
;--- Objektdaten von BOB 1---
ObjektArgs1:
  dc.w 0,0                       ; OLDY,OLDX
  dc.b 0,0                       ; BOBOFF = Normal , Init = 0
  dc.w 100,20                    ; YPos,XPos
  dc.w 8,3,2                     ; Höhe = 8, Wordbreite = 3,
                                   Tiefe = 2
```

```
  dc.l image1                    ; Adresse der BOB-Daten
  dc.l 0,0,0                     ; ShadowMask/SaveBuf-
                                   fer/CollMask = Null
image1:                          ; Imagedaten vom BOB
  dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
  dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
  dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
  dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
;
;--- Objektdaten von BOB 2---
ObjektArgs2:
  dc.w 0,0                       ; OLDY,OLDX
  dc.b 0,0                       ; BOBOFF = Normal , Init = 0
  dc.w 10,200                    ; YPos,XPos
  dc.w 8,3,2                     ; Höhe = 8, Wordbreite = 3,
                                   Tiefe = 2
  dc.l image2                    ; Adresse der BOB-Daten
  dc.l 0,0,0                     ; ShadowMask/SaveBuf-
                                   fer/CollMask = Null
image2:                          ; Imagedaten vom BOB
  dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
  dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
  dc.w
$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0
  dc.w
$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0
;
;--- Objektdaten von BOB 3---
ObjektArgs3:
  dc.w 0,0                       ; OLDY,OLDX
  dc.b 0,0                       ; BOBOFF = Normal , Init = 0
  dc.w 200,80                    ; YPos,XPos
  dc.w 8,3,2                     ; Höhe = 8, Wordbreite = 3,
                                   Tiefe = 2
  dc.l image3                    ; Adresse der BOB-Daten
  dc.l 0,0,0                     ; ShadowMask/SaveBuf-
                                   fer/CollMask = Null
image3:                          ; Imagedaten vom BOB
```

```
  dc.w
$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0
  dc.w
$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0,$0000,$0000,$0
dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
dc.w $ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0,$ffff,$ffff,$0
;
bobtabelle: blk.l 20,0                ; Platz f r 4 Objekte
;
oldirqebene: dc.l 0
intena_buf: dc.w 0
;
irq_change: dc.w 0
;
speed1: dc.w 1,1                      ; Y/X-Speed von Objekt 1
speed2: dc.w 2,2                      ; Y/X-Speed von Objekt 2
speed3: dc.w 2,1                      ; Y/X-Speed von Objekt 3
;
gfxbase: dc.l 0                       ; graphics.basis
gfxname: dc.b "graphics.library",0 ; Libraryname
 even
bitmap_tabelle: blk.l 3,0
bitmapA: blk.b 40,0                   ; BitMap-Struktur A
bitmapB: blk.b 40,0                   ; BitMap-Struktur B
bitmapC: blk.b 40,0                   ; BitMap-Struktur C
rastport: blk.b 100,0                 ; RastPort-Struktur
colortable:
dc.w 0,1000,200,600,500,1040,2340,1234,4000,12,456,876
blk.w 20,100                          ; 32 Farben
spritedatas: blk.l 3,0
copperlist:                           ; ab hier Copperliste
  dc.w $8e,$3081                      ; DIWSTRT
  dc.w $90,$30c1                      ; DIWSTOP
  dc.w $92,$38                        ; DDFSTRT
  dc.w $94,$d0                        ; DDFSTOP
colorpuffer:
  blk.b 128,0                         ; Farbregister
copperpuffer:
```

```
  blk.b 60,0                    ; Screen-Customregister
  dc.w $ff01,$fff4              ; Wait 255
  dc.w $ffdf,$fffe              ; Wait maximale Pos.
  dc.w $0101,$fffe              ; Wait 1 (= 256)
  dc.w $3001,$fffe              ; max. $3801
  dc.w $009c,$8010              ; CopperIRQ auslesen
  dc.l $fffffffe                ; Copperlistende
demotext:
  dc.b "DB-BOBs mit Collision!",0
  even
  END
;--- Listingende ---
```

7.12 Animierte BOBs

Unter Animation versteht man die Bewegung eines sich selbst in unterschiedlichen Bewegungsphasen befindlichen Objektes auf dem Bildschirm. Die Animation kann auf verschiedene Arten ablaufen. Einmal zeichnet man sich alle einzelnen Bewegungsphasen (Sequencen) und läßt diese nacheinander ablaufen. Oder es werden nur die Positionsvariable geändert, also das Objekt bewegt. Die letzte Möglichkeit wäre, beide Systeme miteinander zu verbinden.

Das einfache Ändern der Positionen haben wir ja bereits im vorangegangen Abschnitt programmiert und dürfte deswegen wohl kein Hindernis darstellen. Möchte man jetzt verschiedene Sequencen ablaufen lassen, damit zum Beispiel der Eindruck entstünde, eine Kugel würde sich drehen, so kann dies wie folgt realisiert werden:

Entweder man legt sich für jede einzelne Sequence eine Objekt-Args-Atruktur an und aktiviert diese im richtigen Augenblick für die vorangegangene. Oder es wird für alle Sequencen eines Objektes nur eine ObjektArgs- Struktur angelegt und im richtigen Augenblick wird nur die Image-Adresse entsprechend geändert. Allerdings

müssen dann, je nachdem welche Double-Buffering Methode angewendet wird, alle Sequencen die gleichen Ausmaße besitzen. Wir wollen hier diesmal aber keine Beispielroutinen vorstellen, die uns die Animation abnimmt, sondern diesmal sind sie selbst gefragt. Aber eine kleine Hilfe soll ihnen nicht verwert bleiben:

Am besten sie entwerfen eine Animations-Struktur, die alle nötigen Parameter für die Animation enthält, wie zum Beipiel:

- Animationsgeschwindigkeit
- Bewegungsgeschwindigkeit
- Pausenlänge oder Ausschalten nach Animationsablauf
- Bewegungsrichtung
- Anzahl der Sequencen (ObjektArgs-Strukturen)
- Adressen der einzelnen Sequencen (ObjektArgs-Strukturen)
- usw.

Diese Struktur entwerfen sie für jedes Objekt. Dann fassen sie alle Animations-Strukturen in einer Tabelle zusammen (siehe BOB-Tabelle). Jetzt brauchen sie nur noch eine Routine zu programmieren, der sie die Adresse der Animtions-Tabelle übergeben und diese verwaltet dann die Parameter entsprechend. Diese Routine müssen sie natürlich regelmäßig aufrufen.

Damit die Objekte bzw. entsprechenden Sequencen auch erscheinen, müssen sie noch eine Routine programmieren, die aus der Animations-Tabelle eine fertige BOB-Tabelle erstellt. Diese können sie dann, wie gewohnt verwalten.

Kapitel 8

Konzeptablauf eines Spieles

Das Schlimmste, was sie wohl machen könnten, ist einfach "drauflos" zu programmieren. Denn irgend wann kommt man an einen Punkt, wo das Spiel so komplex wird, das es sich nicht mehr überblicken läßt.

Nehmen wir mal an, sie wollen ein Ballerspiel programmieren. Dazu müßten sie sich die ganzen Startkoordinaten, Namen der Objekte, alle dazugehörigen Strukturen, Kollisionsbedingungen usw. merken. Das wird mit der Zeit viel zu viel. Außerdem sollte ein Außenstehender sich auch in ihrem Listing zurechtfinden; also immer alles gut dokumentieren. Das ist zwar eine Menge Arbeit, sie rechtfertigt sich aber, sobald sich der erste Programmierfehler einnistet.

Zunächst sollte festgelegt werden, um was für eine Art von Spiel es sich handeln soll (Abenteuer, Action, Strategie, Sport usw.). Danach entwickeln sie eine Programm- bzw. Spielidee, die auch machbar ist. Das heißt, eine die sich mit ihren Programmierkenntnissen realisiern läßt. Als dritten Schritt fertigen sie eine Inhaltsangabe an, mit der sie auch weitestgehend alle zu benötigen Parameter auflisten. Am besten sie stellen sich eine Reihe von Fragen, die ihnen so einfallen, wie zum Beispiel:

- Was ist die Aufgabe des Spielers (Spielsinn ?)
- Wie sehen die Feinde, der Hintergrund und die Spielfigur aus ?
- Wieviel Leben besitzt man ?
- Wann verliert man einen Spieler bzw. ein Leben ?
- Wie kann man den Gefahren ausweichen bzw. sich davor schützen ?

- Wie gelangt man in den nächsten Level ?
- Wieviele Level gibt es ?
- Wie und in wieviel Richtungen läßt sich die Figur lenken ?
- Kann man Gegenstände bzw. Hilfsmittel einsammeln ?
- Wie kann man sich verteidigen ?
- Wird die Handlung im Laufe des Spieles schwerer ?
- Wann ist das Spiel beendet ?
- usw.

Nach der Inhaltsangabe wird jetzt eine Liste aller benötigten Hintergrunde und Figuren aufgestellt. Da der eigentliche Spielablauf jetzt soweit klar sein sollte, wird danach der Programmablauf skizziert bzw. beschrieben. Welche Routinen programmiert werden müssen und wie diese zusammenarbeiten. Dabei sollten man auch gleich den Speicher einteilen (Programm, Grafik und Sound).

Die eigentliche Arbeit beginnt jetzt mit der Programmierung der einzelnen Routinen, dem Zeichnen der Figuren und Hintergründe und dem Erstellen der Animation.

Um alle diese Schritte zu bewältigen, benötigt man mindestens zwei bis drei Programme:

1. Assembler, mit dem man programmiert
2. Zeichenprogram (zum Beispiel DeluxePaint)
3. evtl. ein Animationsprogramm (am besten ein eigenes)

Das folgende Schaubbild verdeutlicht nocheinmal alle Schritte in Form einer Kurzbeschreibung:

Von der Idee zum fertigen Spiel:

1. Art des Spiels festlegen
2. Programm bzw. Spielidee entwickeln, die auch machbar ist.
3. Kurze Inhaltsangabe (Fragen beantworten)
4. Auflistung aller benötigten Figuren und Hintergründe
5. Kurz den Programmablauf skizzieren bzw. beschreiben
6. Sound bzw. Titelmusik wählen

7. Skizze der zu programmierenden Routinen erstellen (Wie sie zusammen arbeiten) und Speicher einteilen
8. Die einzelnen Routinen programmieren
9. Figuren bzw. Hintergründe zeichnen
10. Animation erstellen (eftl. mit eigenem Animator), die dann für jede Figur fertig abgespeichert wird.

8.1 Konzept eines Beispielprogrammes

Wir werden in diesem Buch kein Spiel programmieren, da dies viel zu platzraubend ist. Stattdessen werden wir ein Beispielkonzept für ein Ballerspiel (Weltraum) entwerfen.

Das Buch soll ihnen ja nur die nötigen Grundkenntnisse vermitteln, die zur Spieleprogrammierung erforderlich sind. Das eigentliche Spiel müssen sie dann schon selbst erstellen. Die wichtigsten Routinen sind ja im Buch enthalten.

Beispielkonzept eines Weltraumspiels:

1. Art des Spiels festlegen:

Es ist ein Action-Weltraum-Ballerspiel ohne Strategieteil.

2. Programm- bzw. Spielidee entwickeln, die auch machbar ist:

Der Spieler fliegt mit seinem Raumschiff immer über den selben Hintergrund (Sterne, Planeten, Sonnensysteme, Milchstraßen). Oben bzw. unten am Bildschirmrand scrollt in jedem Level eine andere Fassade (Felsstücke, Frackteile...). Dabei muß man versuchen, entgegenkommenden Hindernissen auszuweichen oder sie zu vernichten. Einige Hindernisse fliegen ununterbrochen über die Spielfläche und andere befinden sich solange darauf, bis man sie zerstört hat. Am Ende eines jeden Levels folgt eine Bonusrunde (zu jedem Level eine andere Bonusrunde). Jeder Level wird nachgeladen.

3. Kurze Inhaltsangabe - Fragen beantworten

a) Sinn: So viele Punkte wie möglich zu erreichen. Eintragung in Top-Ten Hi-Scoreliste möglich, welche auch abgespeichert wird

b) Aussehen: Auf Extra-Blättern skizzieren

c) Lebenanzahl wird durch einen Energiestreifen angezeigt. Wird der Spieler getroffen, so wird der Streifen ein Stück kleiner (Feind- bzw. Fassadenberührung). Jedoch muß der Level nicht wieder von vorne begonnen werden. Wenn der Streifen aufgelöst ist, so ist das Spiel beendet.

d) Alle 10000 Punkte wird der Streifen ein wenig größer

e) Schutz: In dem man die Feinde mit verschiedenen Waffen abschießt.

f) Wenn man sich tapfer durch einen Level geschlagen hat, gelangt man automatisch in den nächsten (mit Bonusrunde).

g) Es gibt eine Diskette voll mit Levels, die jeweils nachgeladen werden.

h) Das Raumschiff läßt sich mit dem Joystick in alle acht Richtungen lenken

i) Verteidigung: Am unteren Bildschirmrand werden mehrere Waffen angezeigt, die mit den Funktionstasten anwählbar sind. Jede Waffe hat eine andere Auswirkung. Mit einen Farbstreifen wird die noch vorhandene Energie der dazugehörigen Waffe angezeigt. Wenn die Energie einer Waffe verbraucht ist, dauert es eine Zeit, bis sie sich wieder aufgeladen hat.

4. Auflistung aller benötigten Figuren + Hintergründe

Eine Tabelle anlegen in der die Hindernisart, der Bewegungsablauf, die Anzahl der Objekte eingetragen wird.

Alle anderen Schritte sind jetzt von ihnen abhängig. Für den Programmablauf hilft es, wenn man sich ein paar Schablonen anfertigt (für Collisions-, Animations-, Sound- usw. Routinen). Diese braucht man sich dann nur noch zurechtlegen und fast fertig ist der Programmablauf.

Anhang A

Strukturen

BitMap-Struktur: (Länge = 40 Bytes)

Offset	Typ	Bezeichnung	Funktion
00	Word	BytePerRow	Anzahl Bytes einer Zeile
02	Word	Rows	Anzahl Zeilen der Bitmap
04	Byte	Flags	System-Flags
05	Byte	Depth	Anzahl der Bitmaps (Tiefe)
06	Word	Pad	unbenutzt
08	Long	PlanePtr1	Adresse der 1. Bitmap
12	Long	PlanePtr2	Adresse der 2. Bitmap
16	Long	PlanePtr3	Adresse der 3. Bitmap
20	Long	PlanePtr4	Adresse der 4. Bitmap
24	Long	PlanePtr5	Adresse der 5. Bitmap
28	Long	PlanePtr6	Adresse der 6. Bitmap
32	Long	PlanePtr7	Adresse der 7. Bitmap
36	Long	PlanePtr8	Adresse der 8. Bitmap
40	END		Ende der BitMap-Struktur

'RastPort'-Struktur (Länge = 100 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
000	Long	Layer	Zeiger auf 'Layer'-Struktur
004	Long	BitMap	Zeiger auf 'BitMap'-Struktur
008	Long	AreaPtrn	Zeiger auf das AreaFill-Muster
012	Long	TmpRas	Zeiger auf 'TmpRas'-Struktur
016	Long	AreaInfo	Zeiger auf 'AreaInfo'-Struktur
020	Long	GelsInfo	Zeiger auf 'GelsInfo'-Struktur
024	Byte	Mask	Schreibmaske (Bitmapmaske)
025	Byte	FgPen	Farbregisternummer für Vordergrundfarbe
026	Byte	BgPen	Farbregisternummer der Hintergrundfarbe
027	Byte	AOIPen	Farbregisternummer der AreaFill-Outlinefarbe
028	Byte	DrawMode	ZeichenModi (0=Jam1, 1=Jam2, 2=Complement, 3=Invers) für Routine 'SetDrMd ()'
029	Byte	AreaPtSz	Höhe des AreaFill-Musters (2^n Words))
030	Byte	LinePatCNT	unbenutzt
031	Byte	Dummy	Line Draw Pattern Preshift
032	Word	Flags	verschiedene Kontrollbits (FirstDot etc.)
034	Word	LinePtrn	16 Bits für Linienmuster
036	Word	CP_X	x-Position des Grafikcursors
038	Word	CP_Y	y-Position des Grafikcursors
040	Byte	Minterms(8)	8 x 1 Byte für Minterms (Funktion unbekannt)
048	Word	PenWidth	Cursorbreite
050	Word	PenHeight	Cursorhöhe
052	Long	TextFont	Zeiger auf 'TextFont'-Struktur
056	Byte	AlgoStyle	Zeichensatzmodus (Style-Flag)
057	Byte	TxFlags	textspezifische Flags

058	Word	TxHeight	Höhe des Zeichensatzes
060	Word	TxWidth	durchschnittliche Zeichenbreite
062	Word	TxBaseLine	Texthöhe ohne Unterlängen
064	Word	TxSpacing	Zeichenabstand
066	Long	RP-User	Zeiger auf evtuelle Userdaten (unwichtig!)
070	Word		7 Words reserviert
084	Long		2 Longs reserviert
092	Byte		8 Bytes reserviert
100	End		Ende der Datenstruktur

SoundTable -Struktur: (Länge = 16 Bytes)

Offset	Typ	Bezeichnung	Funktion
00	Long	Parameter0Ptr	Zeiger auf Parameterstruktur von Kanal 0
04	Long	Parameter1Ptr	Zeiger auf Parameterstruktur von Kanal 1
08	Long	Parameter2Ptr	Zeiger auf Parameterstruktur von Kanal 2
12	Long	Parameter3Ptr	Zeiger auf Parameterstruktur von Kanal 3
16	END		Ende der SoundTable-Struktur

Die Zahl hinter Parameter gibt den Soundkanal an, über dem der Sample abgespielt wird. Soll ein Kanal nicht benutzt werden, muß das Langword (Long) an der entsprechenden Stelle auf Null gesetzt werden.

ParameterXPtr -Struktur: (Länge = 12 Bytes)

Offset	Typ	Bezeichnung	Funktion
00	Long	SampleData	Anfangsadresse der Sampledaten
04	Long	Samplelänge	Anzahl Bytes der Sampledaten
08	Word	Periodendauer	Abtastrate (Tonhöhe) vom Sample
10	Word	Volume	Lautstärke vom Sample (0 bis 64)
12	END		Ende der ParameterXPtr-Struktur

BlitterArgs -Struktur: (Länge = 22 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
00	Long	Quelle	Anfangsadresse des Quelldatenbereichs, welcher verschoben werden soll.
04	Long	Ziel	Anfangsadresse des Zielbereichs, zu dem die Daten, auf die Quelle zeigt, hinverschoben werden sollen.
08	Word	QModulo	Modulowert der Quelle in Bytes
10	Word	ZModulo	Modulowert des Ziels in Bytes
12	Word	WBreite	Breite der Grafik, welche verschoben werden soll, in Words.
14	Word	Höhe	Höhe der Grafik in Zeilen

16	Byte	Art	Art des Datentransfers: 1 = nur gesetzte Bits werden kopiert 0 = alle Bits werden kopiert
17	Byte	QShift	Anzahl Punkte die der Quelldatenbereich vor der Ausgabe nach rechts geschoben werden soll. Werte zwischen 0 und 15 sind erlaubt.
18	Word	FirstMask	erste Maske der Quelle (16 Bits)
20	Word	LastMask	letzte Maske der Quelle (16 Bits)
22	END		Ende der BlitterArgs-Struktur

ObjektArgs -Struktur: (Länge = 32 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
00	Word	OldY	alte Y-Position (Intern für Routine)
02	Word	OldX	alte X-Position (Intern für Routine)
04	Byte	BOBOff	Wert, Funktion: 0: Normal (führt Schritte 3/4/2 aus) 1: BOB wird ausgeschaltet (Schritt 3) 2: Schreibe nur BOB-Daten in BitMap 3: Hintergrunddaten zurückschreiben 4: Hintergrunddaten lesen
05	Byte	Init	Variable für Routine

06	Word	YPos	Vertikale Position vom Objekt
08	Word	XPos	Horizontale Position vom Objekt
10	Word	Height	Höhe vom Objekt in Zeilen
12	Word	WordWidth	Breite vom Objekt in Words
14	Word	Depth	Anzahl Bitmaps (Planes) vom Objekt
16	Long	Image	Zeiger auf Grafikdaten vom Objekt
20	Long	ShadowMask	Zeiger auf Schattenmaske (Wichtig)
24	Long	SaveBuffer	Zeiger auf Hintergrundpuffer
28	Long	CollMask	Zeiger auf Collisionsmaske
32		END	Ende der ObjektArgs-Struktur

BOB-Tabelle: (Länge mindestens 6 Bytes)

Offset	Typ	Bezeichnung	Beschreibung
00	Word	Count	Anzahl ObjektArgs-Strukturen
02	Long	ObjektArgs0	Zeiger auf ObjektArgs-Struktur von BOB 0
06	Long	ObjektArgs1	Zeiger auf ObjektArgs-Struktur von BOB 1

.
.
.
so oft wie in Count angegeben ist.

In Count wird angegeben, wieviel ObjektArgs-Strukturen die BOB-Tabelle enthält. Ab Offset zwei stehen die Anfangsadressen der ObjektArgs-Strukturen. Dabei besitzt das zuletzt eingetragene Objekt die höchste Priorität, verdeckt alle dahinterliegenden Objekte.

Aufbau der BOB-Headerdatei:

Offset	Typ	Bezeichnung	Beschreibung
00	Long	TBOB	Kennzeichnung der Headerdatei
04	Long	V1.0	Transformerversion
08	Long	Länge	Länge der gesammten BOB-Datei
12	Long	HEAD	4-Byte Kennzeichnung
16	Word	Höhe	Höhe des BOBs in Zeilen
18	Word	WordBreite	Breite des BOBs in Words (1 Word = 16 Bit)
20	Word	Breite	Breite des BOBs in Pixeln
22	Word	Tiefe	Anzahl Bitmaps des BOBs
24	Word	Frei	unwichtig
26	Word	YPos	Y-Position vom BOB
28	Word	XPos	X-Position vom BOB
30	Word	Frei	unwichtig
32	Long	BODY	Kennzeichnung der Imagedaten
36	Long	Grafiklänge	Länge der eigentlichen Grafikdaten
40	Byte		ab hier liegen die eigenlichen Grafikdaten

Anhang B

Routinen

Routinenliste:

Disable ()
Enable ()
AllocMem (Größe,Code) (D0,D1)
FreeMem (Adresse,Größe) (A1,D0)
AvailMem (Code) (D1)
AllocAbs (Größe,Adresse) (D0,A1)
Open (Filename,Modus) (D1,D2)
Read (Fileadresse,Speicher,Länge) (D1,D2,D3)
Close (Fileadresse) (D1)
InitBitMap (Tiefe,Breite,Höhe,Bedingung,BitMap) (D0-D3,A0)
ClearBitMap (BitMap) (A0)
PrintIffPic (BitMap,IffPic,ColorTable) (A0-A2)
GetBackMask (ObjektArgs,BitMap (A0,A1)
FreeBackMask (ObjektArgs) (A1)
InitRastPort (RastPort) (A1)
Text (RastPort,Text,Anzahl) (A1,A0,D0)
TextLength (RastPort,Text,Anzahl) (A1,A0,D0)
Move (RastPort,X,Y) (A1,D0,D1)
PrintText (Text,RastPort,Grafikbasis,X,Y) (A0,A1,A6,D0,D1)
FillBitMap (Muster,BitMap (D0,A1)
InitColor (ColorTable,Colorpuffer) (A0,A1)
InitCopperMap (BitMap,Copperpuffer,Modus) (A0,A1,D0)
PlaySample (Soundtabelle) (A0)
CopyGrafik (BlitterArgs) (A0)
CopyBitMap (QuellBitMap,ZielBitMap) (A0,A1)
PrintObjekt (ObjektArgs,BitMap) (A0,po_bitmap)

InitMask (ObjektArgs,Bedingung) (A0,D0)
GetSaveBuffer (ObjektArgs,BitMap) (A0,A1)
FreeMask (ObjektArgs) (A0)
FreeSaveBuffer (ObjektArgs,BitMap) (A0,A1)
WriteBackgroundsOnly (BOB-Tabelle) (printbob_tabelle)
ReadBackgroundsOnly (BOB-Tabelle) (printbob_tabelle)
WriteBOBsOnly (BOB-Tabelle) (printbob_tabelle)
Collision (ObjektArgs1,ObjektArgs2) (A0,A1)

Library: exec.library
Routine: Disable ()
Offset : -120 = -$78
Parameter: keine
Erklärung: Diese Routine schaltet alle Interrupts aus.

Library: exec.library
Routine: Enable ()
Offset: -126 = -$7e
Parameter: keine
Erklärung: Diese Routine schaltet alle Interrupts die mit der Routine Disable () ausgeschaltet wurden, wieder ein.

Routine: AllocMem (D0,D1) (Größe,Code)
Library: exec.library
Offset: -198 = -$c6
Parameter: D0 = Gibt die Größe des Speichers an der reserviert werden soll.
D1 = Bedingungs-Code um was für eine Art von Speicher es sich handeln soll (siehe oben).
Rückgabe: in D0 = Adresse des reservierten Speicherbereichs. Wenn eine Null zurückgegeben wird, konnte der Speicher nicht reserviert werden.
Erklärung: Diese Routine reserviert einen unbestimmten Speicherbereich von angegebener Größe und Bedingung.

Routine: FreeMem (A1,D0) (Adresse,Größe)
Library: exec.library
Offset: -210 = -$d2
Parameter: A1 = Adresse des Speicherblocks welcher freigegeben werden soll.
D0 = Gibt die Größe des Speichers an, den man freigeben möchte
Erklärung: Diese Routine gibt den Speicherbereich wieder frei, welcher zuvor mit der Routine AllocMem () reserviert worden ist.

Routine: AvailMem (D1) (Code)
Library: exec.library
Offset: -216 = -$d8
Parameter: D1 = Bedingungs-Code des Speichers, welcher untersucht werden soll.
Rückgabe: in D0 = Größe des Speichers, der noch zur Verfügung steht.
Erklärung: Diese Routine gibt die Größe des Speichers an, der noch zur Verfügung steht, bezogen auf die Bedingung.

Routine: AllocAbs (D0,A1) (Größe,Adresse)
Library: exec.library
Offset: -204 = -$cc
Parameter: D0 = Größe des Speicherblocks der reserviert werden soll
A1 = Anfangsadresse des Speicherblocks, welcher reserviert werden soll.
Rückgabe: in D0 = Die Adresse, welcher vorher übergeben wurde oder eine Null, wenn der Speicherblock nicht zugewiesen werden konnte.
Erklärung: Diese Routine versucht einen Speicherblock an der angegebenen Adresse und von angegebener Größe zu reservieren. Bei Mißlingen wird in D0 eine Null als Fehlermeldung zurückgegeben. Ansonsten die Adresse die zuvor auch übergeben wurde.

Routine: OPEN (D1,D2) (Modus,Filename)
Library: dos.library
Offset: -30 = -$1e
Parameter: D1 = Wert 1005 für Lesen der Daten von Disk
D1 = Wert 1006 für Schreiben auf Disk
D2 = Adresse des Filenames welcher mit Null endet
Rückgabe: in D0 = Fileadresse des Files auf Diskette oder Null, wenn File sich nicht auf Diskette befindet.
Erklärung: Diese Routine gibt die Fileadresse zurück um über ihr weitere Funktionen des DOS steuern zu können (siehe READ).

Routine: READ (D1,D2,D3) (Fileadresse,Datenpuffer,Länge)
Library: dos.library
Offset: -42 = -$2a
Parameter: D1 = Adresse des Files auf Diskette. Liefert uns die Routine OPEN () in D0.
D2 = Adresse des Datenpuffers, in dem die Daten abgelegt bzw. von dem sie gelesen werden sollen.
D3 = Anzahl Bytes die geladen bzw. geschrieben werden sollen.
Rückgabe: in D0 = Anzahl Daten die tatsächlich gelesen bzw. geschrieben wurden. Normalerweise ist diese gleich der in Länge übergeben wurde.
Erklärung: Diese Routine lädt eine bestimmte Anzahl Daten von Diskette in den Speicher bzw. vom Speicher auf die Diskette.

Routine: CLOSE (D1) (Fileadresse)
Library: dos.library
Offset: -36 = -$24
Parameter: D1 = Fileadresse des Files, welches zuvor mit OPEN () geöffnet wurde.
Erklärung: Diese Routine schließt ein File wieder, damit später auch ordnungsgemäß wieder darauf zugegriffen werden kann.

Routine: InitBitMap (A0,D0,D1,D2,D3)
(BitMap,Depth,Width,Height,Memory)
Parameter: A0 = Zeiger auf BitMap-Struktur (Puffer von 40 Bytes)
D0 = Anzahl Bitmaps (Tiefe)
D1 = Breite in Pixel - Achtung muß durch 16 teilbar sein
D2 = Höhe der Grafik
D3 = 1, dann wird auch gleich Speicher für PlanePtr reserviert und Adressen in Struktur eingetragen
D3 = 0, kein Speicher wird für PlanePtr reserviert
Rückgabe: D0 = 0 , alles OK.
D0 = -1, Breite konnte nicht durch 16 geteilt werden
D0 = -2, Speicher für PlanePtr konnte nicht reserviert werden
Erklärung: Initialisiert eine BitMap-Struktur mit den entsprechenden Werten, genau wie die gleichnamige System-Routine. Nur ist diese Routine schneller und außerdem kann man, wenn man D3 = 1 setzt, automatisch gleich Speicher für die BitMap-Pointer reservieren lassen, diese werden sogleich in die Struktur gespeichert.

Routine: ClearBitMap (A0) (BitMap)
Parameter: A0 = Adresse der BitMap-Struktur
Erklärung: Gibt den mit InitBitMap () reservierten Speicher der PlanePtr wieder zurück und löscht diese Zeiger danach.

Routine: PrintIffPic (A0,A1,A2) (BitMap,IffPic,ColorTable)
Parameter: A0 = Adresse der BitMap-Struktur in welcher das Bild dargestellt werden soll.
A1 = Adresse des Iff-Bildes im Speicher.
A2 = Adresse eines Puffers von 64 Bytes, in dem die Farben vom Iff-Pic gespeichert werden.
Erklärung: Diese Routine stellt ein Iff-Bild in der gewünschten BitMap da. Dabei ist zu beachten, das die Breite, Höhe und Tiefe der BitMap-Struktur genausogroß sind wie das IFF-Bild.

Routine: GetBackMask (A0,A1) (BitMap,ObjektArgs)
Parameter: A0 = Zeiger auf BitMap-Struktur von welcher die Schattenmaske angelegt werden soll.
A1 = Anfangsadresse eines 32 Bytes großen Puffers. Er enthält die ObjektArgs-Struktur, welche im Kapitel 7 beschrieben wird.
Rückgabe: D0 = 0 , Funktion erfolgreich ausgeführt.
D0 = -1, es konnte nicht genug Speicher für die Schattenmaske reserviert werden.
Erklärung: Füllt die in A1 übergebene ObjektArgs-Struktur mit den wichtigsten Werten und reserviert Speicher für die Schattenmaske (ShadowMask/CollMask). Diese ObjektArgs-Struktur kann dann ganz normal mit der Collision () Routine (Kapitel 7) zur Collision mit anderen Objekten herangezogen werden.
Tip: Wenn man die BitMap-Pointer und die Tiefe in der BitMap-Struktur vorher entsprechend manipuliert, kann man sogar Collisionen nur zwischen bestimmten Farbregistern zulassen.

Routine: FreeBackMask (A1) (ObjektArgs)
Parameter: A1 = Adresse der ObjektArgs-Struktur (32 Bytes Puffer) in der die Schattenmaske für den Hintergrund steht.
Erklärung: Gibt den ShadowMask-Puffer, der mit der Routine "GetBackMask ()" für die Hintergrundmaske reserviert worden ist, wieder an das System zurück.

Routine: InitRastPort (A1) (RastPort)
Offset: -198 = -$c6
Library: graphics.library
Parameter: A1 = Adresse von einem 100 Bytes großen Puffer.
Erklärung: Diese Routine füllt die von uns angelegte RastPort-Struktur mit den nötigsten Startwerten. Diese Routine muß noch mit der BitMap-Struktur verbunden werden. Sonst kann kein Text dargestellt werden.

Routine: Text (String,RastPort,Count) (A0,A1,D0)
Offset: -60 = -$3c
Library: graphics.library
Parameter: A0 = Anfangsadresse des Textes
A1 = Anfangsadresse der RastPort-Struktur
D0 = Anzahl der Buchstaben, die ausgegeben werden sollen
Erklärung: Diese Routine gibt einen Text im angegebenen RastPort aus.

Routine: TextLength (String,RastPort,Count) (A0,A1,D0)
Offset: -54 = -$36
Library: graphics.library
Parameter: siehe Funktion Text ().
Rückgabe: in D0 = Länge des Textes in Pixeln.
Erklärung: Diese Routine gibt für den angegebenen Text die Textlänge in Pixeln, im Datenregister D0 zurück.

Routine: Move (RastPort,X,Y) (A1,D0,D1)
Offset: -240 = -$f0
Library: graphics.library
Parameter: A1 = Anfangsadresse der RastPort-Struktur
D0 = X-Position (horizontale) des Textes
D1 = Y-Position (vertikale) des Textes
Erklärung: Setzt den Grafikcursor auf die angegebenen Koordinaten. Danach kann dann zum Beispiel mit der Funktion "Text ()", ab diesen Positionen Text ausgegeben werden.

Routine: PrintText (A0,A1,A6,D0,D1)
(Text,RastPort,GraphicsBasis,X,Y)
Parameter: A0 = Zeiger auf Text, welcher mit Null endet
A1 = Adresse der RastPort-Struktur
A6 = Grafikbasisadresse
D0 = X-Position des Textes
D1 = Y-Position des Textes
Erklärung: Gibt einen Text auf dem Bildschirm aus.

Routine: FillBitMap (D0,A1) (Muster,BitMap)
Parameter: D0 = Füllmuster (16 Bit bzw. 1 Word groß), mit dem die BitMap gefüllt werden soll.
A1 = Adresse der BitMap-Struktur
Erklärung: Diese Routine füllt eine Bitmap auf die A1 zeigt, mit einem Wordgroßen Muster, welches in D0 übergeben wird. Die BitMap darf maximal die Ausmaße 1024 * 1024 haben.

Routine: InitColor (A0,A1) (ColorTable,Copperpuffer)
Parameter: A0 = Zeiger auf einen Puffer von 32 Words in dem die Farbwerte für die 32 Farbregister stehen.
A1 = Zeiger auf einen 128 Bytes großen Puffer, innerhalb der Copperliste, in dem die 32 Farben kopiert werden.
Erklärung: Kopiert die 32 Farben, auf die ColorTable zeigt, in den 128 Bytes großen Copperpuffer und erzeugt auch die dazu gehörigen MOVE-Befehle.
Achtung! - Diese Routine beim erstenmal, vor dem aktivieren der Copperliste aufrufen. Damit die Farbregister ($180 bis ...) schon einmal in der Copperliste enthalten sind. Danach kann man diese Routine jederzeit wieder aufrufen und damit die Farben ändern.

Routine: InitCopperMap (A0,A1,D0)
(BitMap,Copperpuffer,Modus)
Parameter: A0 = Zeiger auf BitMap-Struktur
A1 = Zeiger auf einen 60 Bytes großen Puffer, innerhalb der Copperliste, in dem die Register kopiert werden.
D0 = Grafik-Modus:
= $8000 für Hires
= $800 für HAM (Hold and Modify)
= $400 für DualPlayField
= $4 für Interlace
= $0 für Normal
Erklärung: Setzt die wichtigsten Startwerte zum aktivieren der Grafik. Es werden die Customregister BPL1MOD, BPL2MOD, BPLCON0 und die BitMap-Pointer mit den entsprechenden Werten gefüllt. Diese Register werden dann in den 60 Bytes großen Puffer, innerhalb der Copperliste, kopiert.
Achtung! - siehe Achtung! bei InitColor ().
Achtung! - Wir verwenden in diesem Buch nur den Normalmodus, weil die anderen einer weiteren Erklärung bedürfen. Sie sind für die Spieleprogrammierung nicht besonders geeignet.

Routine: PlaySample (A0) (SoundTable)
Parameter: A0 = Zeiger auf SoundTable-Struktur
Erklärung: Spielt einen Sample auf dem gewünschten Kanal einmal ab.
Achtung! Diese Routine wartet nicht, bis der Sample zuende gespielt wurde. Man darf deswegen nicht zu schnell Samples auf dem selben Kanal hintereinander abspielen.

Routine: CopyGrafik (A0) (BlitterArgs)
Parameter: A0 = Zeiger auf Blitterargumentenliste, welche alle nötigen Informationen für den Blitter enthält.
Erklärung: Kopiert Daten von einem Quellbereich zu einem Zielbereich.

Routine: CopyBitMap (A0,A1) (Quell-BitMap,Ziel-BitMap)
Parameter: A0 = Adresse der Quell-BitMap-Struktur, welche verschoben werden soll.
A1 = Adresse der Ziel-BitMap-Struktur, wo die Quell-BitMap-Struktur hinverschoben werden soll.
Erklärung: Diese Routine kopiert die BitMap-Struktur auf die A0 (Quelle) zeigt, in die BitMap-Struktur auf die A1 (Ziel) zeigt. Die BitMap darf maximal 1024 Pixel breit und 1024 Zeilen hoch sein.

Routine: FillBitMap (D0,A1) (Muster,BitMap)
Parameter: D0 = Füllmuster (16 Bit bzw. 1 Word groß) mit dem die BitMap gefüllt werden soll.
A1 = Adresse der BitMap-Struktur, welche mit dem Muster gefüllt werden soll.
Erklärung: Diese Routine füllt eine BitMap-Struktur auf die A1 zeigt, mit einem 16-Bit großen Muster, welches im Datenregister D0 übergeben wird. Die Ausmaße sind die selben wie bei CopyBitMap ().

Routine: PrintObjekt (A0,po_bitmap) (ObjektArgs,BitMap)
Parameter: A0 = Adresse der ObjektArgs-Struktur, welche alle nötigen Parameter enthält.
po_bitmap = Ist eine feste Variable im dazugehörigen Listing. Es enthält die Adresse der BitMap-Struktur in der das Objekt dargestellt werden soll.
Achtung! - Das letzte Word einer Grafikzeile muß immer Null sein.

Achtung! - Wenn man einen Hintergrund zurückschreibt, ohne vorher einen gerettet zu haben, wird die Hintergrundgrafik trotzdem nicht zerstört.

Routine: InitMask (A0,D0) (ObjektArgs,Bedingung)
Parameter: A0 = Adresse der ObjektArgs-Struktur
D0 = 0 - ShadowMask-Puffer wird nicht reserviert, sondern muß vorher schon reserviert worden sein. Es wird nur die Schattenmaske erzeugt und in diesem Puffer abgelegt.
D0 = 1 - ShadowMask-Puffer wird automatisch reserviert und Schattenmaske dort abgelegt.
D0 = 2 - Genau wie "1", nur wird zusätzlich der Coll-Mask-Zeiger gleich dem ShadowMask-Zeiger gesetzt.
Rückgabe: D0 = 0, alles OK!
D0 = -1, nicht genug Speicher vorhanden.
Erklärung: Initialisiert den ShadowMask-Puffer mit der entsprechenden Schattenmaske.

Routine: GetSaveBuffer (A0,A1) (ObjektArgs,BitMap)
Parameter: A0 = Anfangsadresse der ObjektArgs-Struktur
A1 = Anfangsadresse der BitMap-Struktur
Rückgabe: D0 = 0, dann ist alles in Ordnung!
D0 =-1, dann ist nicht genug Speicher vohanden!
Erklärung: Reserviert Speicher für die Hintergrundspeicherung und schreibt die Anfangsadresse in den SaveBuffer -Offset.

Routine: FreeMask (A0) (ObjektArgs)
Parameter: A0 = Anfangsadresse der ObjektArgs-Struktur
Erklärung: Gibt den ShadowMask-Speicher, der mit InitMask () für die Schattenmaske reserviert worden ist, wieder zurück.

Routine: FreeSaveBuffer (A0,A1) (ObjektArgs,BitMap)
Parameter: A0 = Zeiger auf ObjektArgs-Struktur
A1 = Zeiger auf BitMap-Struktur
Erklärung: Gibt den Speicher, auf den SaveBuffer zeigt, wieder frei.

Routine: WriteBackgroundsOnly (printbob_tabelle) (BOB-Tabelle)
Parameter: printbob_tabelle = Adresse der BOB-Tabelle, in der alle BOBs enthalten sind.
Erklärung: Schreibt den Hintergrund von allen BOBs an den alten Positionen in die BitMap zurück.

Routine: ReadBackgroundsOnly (printbob_tabelle) (BOB-Tabelle)
Parameter: siehe WriteBackgroundsOnly ()
Erklärung: Kopiert den Hintergrund von allen BOBs ab den neuen Positionen aus der BitMap in den SaveBuffer-Speicher.

Routine: WriteBobsOnly (printbob_tabelle) (BOB-Tabelle)
Parameter: siehe WriteBackgroundsOnly ()
Erklärung: Kopiert die BOB-Imagedaten von allen BOBs in die BitMap (Hintergrund).

Routine: Collision (A0,A1) (ObjektArgs1,ObjektArgs2)
Parameter: A0 = Adresse der ObjektArgs-Struktur von BOB 1
A1 = Adresse der ObjektArgs-Struktur von BOB 2
Rückgabe: in D0 = 0, dann keine Collision
D0 = 1, dann liegt eine Collision vor
Erklärung: Diese Routine testet eine Collision zwischen zwei Objekten. Wenn eine Collision vorliegt, wird in D0 eine 1 zurückgegeben, andernfalls eine 0. Testet auch Collisionen zwischen ausgeschalteten und sichtbaren BOBs - ist nicht von einer BitMap abhängig.

Anhang C

Programme der Diskette

Auf der beiliegenden Diskette sind alle in diesem Buch aufgeführten Programme, Listings und Routinen. Sie befinden sich in drei Schubladen (Directories), welche sich wie folgt aufteilen:

- Listings
- Programme
- Routinen

In der Schublade Listings sind, wie der Name schon sagt, alle im Buch enthaltenen fünf Listings. In der Schublade Programme sind alle zu den Listings gehörenden lauffähigen Programme. Diese können im CLI gestartet werden. Die Schublade Routinen enthält alle Listings, zu den im Buch beschriebenen Routinen.

Index

Spiele-Programmierung in Assembler

Wie man realistische Spiele selbst programmiert

WICHTIGE MERKMALE:

- Dank der hervorragenden grafischen Eigenschaften des Amiga kann man selbst Spiele programmieren, die absolut realistisch wirken. Das Buch – von vielen Programmier-Freaks erwartet – enthält unter anderem eine Sammlung von Grafik-Routinen, die über bestimmte Parameter die Hardware ansteuern. Deshalb sind keine speziellen Hardware-Kenntnisse erforderlich.
- Das Programmieren von Spielen wird ebenso leicht gemacht wie das Programmieren mit System-Routinen, allerdings mit einem viel größeren Geschwindigkeitsvorteil. Vom Buch profitieren nicht nur Profis, sondern auch Amateure.
- Alle im Buch enthaltenen Routinen stammen aus dem Public Domain-Bereich und können daher frei in eigene Programme eingebaut werden.

AUS DEM INHALT:

- Interrupts:
  - Copper-IRQ
  - Raster-IRQ
- Speicherverwaltung:
  - unbestimmter Speicher
  - bestimmter Speicher
- DOS-System:
  - Daten laden
  - Daten abspeichern
- Darstellung von Bildern:
  - Bitmap
  - IFF-Bilder laden
  - der RastPort
  - Textausgabe
  - der Copper
- Joystick-Abfrage
- Geräuscherzeugung
- Grafik-Hardwareprogrammierung:
  - der Blitter
  - BOBs – Blitter-Objekte
  - IFF-Brushes umwandeln
  - Kollisionen zwischen BOBs
- Konzeptablauf eines Spiels

ISBN 3-928480-01-4
Bestell-Nr. B-511
inkl. Diskette
DM 59,-